Panard Vision: Portable Real-time 3D Engine Documentation
For API 0.98b
Doc revision: 0.6

Last Updated: 26/06/1998

Contact, comments, corrections:
Email: mailto:olivier.brunet@capway.com
Http: http://www.inforoute.capway.com/brunet3/ (Panard Vision home)

You should 'read' the PVISION.H file too, because it is more detailed than here and have some useful comments in it. You should look into the samples accompanying PV for some fancy tricks.

Index

BASICS

Introduction

Starting with 0.98, Panard Vision offers 2 world management mechanisms. The classical one, mesh oriented which encapsulates the following entities:

- Cameras
- Lights
- Meshes
- Materials
- Mirrors
- Worlds

And the more direct one, derived from the OpenGL approach, where you specify per-vertex information. This second mechanism is treated in the *Panard Primitives* section and in the samples.

All the Panard Vision operation are driven by flags and working modes, set by *PV_SetMode()* allowing for efficient, simple and scalable performance management. PV uses a material abstraction to group all faces attributes in an efficient way.

This doc is not an introduction to 3D. You should not be a beginner to read and understand this doc, and what PV can do.

All rendering calls should be enclosed between a *PV_BeginFrame()* call and a *PV_EndFrame()* call. You can mix either mesh oriented API and Panard Primitives between.

Basic types

A byte is 8 bits.
A word is 16 bits.
A dword is 32 bits.
A *Matrix* is an array (3x3) of float
A UPVDx type, is an unsigned x bits integer. Defined types are UPVD8, UPVD16, UPVD32.
A PVDx type, is a signed x bits integer. Defined types are PVD8, PVD16, PVD32.
A PVFLAGS is a 32-bit integer.
All strings are zero terminated and case sensitive when searched.

Basic structures

Struct PVPoint

This structure defines a vertex in 3D space. Fields are :

- *xf* : X coordinates, float.
- *yf* : Y coordinates, float.
- *zf :* Z coordinates, float.

Struct PVRGB

This structure defines an RGB color (24 bits). Fields are:

- *r* : Red component, unsigned byte.
- *g :* Green component, unsigned byte.
- *b :* Blue component, unsigned byte.

Struct PVRGBF

This structure defines an RGB color (24 bits). Fields are:

- *r* : Red component, float.
- *g :* Green component, float.
- *b :* Blue component, float.

- *a :* Alpha component, float. This component is here only to serve user routines. PV ignores it.

Each field is between 0 and 1.

Structure PVQuat

This struct defines a quaternion. Fields are *x,y,z,w* which are the coordinates for the quaternion.

Structure PVPlane

This structure defines a plane in 3D space with the equation $ax+by+cz+d=0$.

- *d :* *d* parameter of the equation.
- *Normal* : *PVPoint* holding *a, b, c* parameters of the equation.

Structure PVPoly

This structure defines ordering of vertex in a polygon.

- *NbrVertices* : Number of vertices in the polygon, up to MAX_VERTICES_PER_POLY vertices are allowed.
- *Vertices[MAX_VERTICES_PER_POLY]* : array of index for each vertex.

Structure PVFogInfo

This structure describes fog data. Not all the fog types are available everywhere.

- *Color* : PVRGB color indicating fog color.
- *Density* : float paraeter of the fog equations below.
- *Start,End :* Corresponding parameters i n the fog equations below.
- *Type* : One of the following value :

PVF_NONE	No fog
PVF_LINEAR	The fog effect intensifies linearly between the start and end points, according to the following formula: F=(end-z)/(end-start)
PVF_EXP	The fog effect intensifies exponentially, according to the following formula: $F=e^{-(density*z)}$
PVF_EXP2	The fog effect intensifies exponentially with the square of the distance, according to the following formula: $F=e^{-(density*z)^2}$

Panard Vision Terminology

Name	Description
Vertex	A point in 3D space, defined with (X,Y,Z) coordinates
Face	A **planar** and **convex** surface made by connecting vertices. A face in Panard Vision may have up to MAX_VERTICES_PER_POLY (defined in config.h) vertices. A face has a material associated with it, which tells the renderer how to draw it.
Mesh	Defines a 3D shape. A mesh is an association between vertices, faces and mapping coordinates. This is the base brick when using PV in non-immediate modes. Meshes can be organized as a hierarchical tree inside a world.
Camera	A viewing position from within the 3D world. A camera contains the field of view, and far and near clipping plane information. A camera can be rotated, or it can possess source and target locations for pointing towards another location. The camera holds viewing frustum (and user defined user clipplanes) information too. Hence defining the 2D Viewport as well. You can have any number of Camera, but only one is active on a given world at a given time.
Light	Provides illumination for shaded rendering methods. A light can have either finite or infinite distance.
Texture	Used to add detail to a face. Any bitmap can be used as a texture.
Material	Defines how a face will look when rendered. Material attributes include color, texture, culling, Zbuffer infos, mipmapping infos ....
World	A world is an association between Meshes, Lights, Materials, Mirrors and a camera. This is what you render in non-immediate modes. PV use the world abstraction to get coherence information on your scene to speed up rendering.
Portal	A link between 2 mesh of a world. When a portal is visible from a a point of view, the mesh connected to it is visible too. See the portal section for more infos.
Mirror	A mirror is a planar polygon, which reflects what is in front of him. A mirror can't reflect what sees another mirror
Texel	A pixel in a texture. A texel can cover many pixels on screen.
Matrix	A matrix is an array of floating point numbers used to define 3D rotations. Matrices are 3x3.

Coordinate Systems

Every mesh is defined by a pivot point, a position in world, 3 scale factors (one for each axis) and a rotation matrix. Each vertex of a mesh is rotated around the pivot point according to the rotation matrix, scaled, and then translated by the world position vector. The vertex obtained by these operations is said expressed in world coordinates.
Then this vertex is transformed and translated by the camera rotation matrix and position vector. This gives the view-space coordinates of the vertex.
Finally the vertex is projected (screen space coordinated) and rendered.
The X-axis increase from left to right.
The Y-axis increase from top to bottom.
The Z-axis increase from far to near.

Error handling and return values of API calls

Error reporting in Panard Vision is simple and more or less efficient ☺. When a function reports an error, it uses an error code (which is the return value of the function), the following error codes may be returned (names are self-explanatory):

COOL	This is no error, you're lucky
FILE_IOERROR	
ALREADY_ASSIGNED	
NO_MEMORY	
BIZAR_ERROR	This is an error, but a not identified one
ARG_INVALID	
ITEM_NOT_FOUND	
MATERIAL_NOT_REGISTERED	
INVALID_CAM	
INVALID_TEXTURE	
NO_MATERIALS	
ALREADY_IN_HIERARCHY	You tried to add a mesh to a world, but this mesh is already owned by a world.
MESH_HAS_CHILD	You tried to add a mesh which has children to a world.
NOT_AVAILABLE	The data you requested are not available.
NO_ACCELSUPPORT	Hardware acceleration not available
ACCEL_NO_MEMORY	No more memory on hardware to store textures
ACCEL_FUNCTION_UNSUPPORTED	Function not supported by the hardware driver
NO_HARDWARE_SET	Hardware function called without setting a driver
INVALID_DRIVER	You tried to set an ill-designed driver
COLLIDE_YES	2 meshes are colliding
COLLIDE_NO	
NO_COLLISIONINFO	No collision tree built

In some cases, functions do not return integers but pointers. In case of errors they return the value NULL.

General usage

Panard Vision uses a lot of structures and global variables. Avoid directly changing the members of structures if API functions exist to do the job. By example do not set the *PV_Mode* variable directly, use *PV_SetMode()* instead. You have been warned, wild modifications could be sources of ……. Huh ….. strange results.

Memory management

All objects should be created by their associated Create function (i.e. create a mesh with *PV_SimpleCreateMesh()*), and destroyed using the associated Kill function (*PV_KillMesh()*). These functions use a reference counting mechanism that allows for multiple references for the same object. You can, for instance, add a light to several Worlds objects. The light will only be freed when the last world is destroyed. That's why the kill functions do not unconditionally free memory when called on an object. They check the reference counter of that object and destroys it only if it is zero.

The PVWorld object is a special one since it's a container for the others. You have several functions to « add » another object (mesh or light for instance) to a world. Once the object is added, the PVWorld object takes control over it, and that's the PVWorld object that will call the Kill functions for each objects. The consequence of this is that you have to create your mesh but you MUST not destroy them, instead you must destroy the world they live in.
That's true for every objects but PVCamera, camera can be shared by worlds but are not linked to them, so you have to create cameras and you have to kill them too.

When you remove an object from a world container (with the provided *PV_Removexxxx()* functions), he is destroyed.

Important notes about malloc(), free() and C runtimes

The Panard Vision DLL is compiled with the DLL version of the C runtime. Hence, you should link your application with the shared version of the C runtime too. Because it's a bad idea to make a malloc with one runtime (static by example) and the corresponding free by another (the DLL one). It may work but could be very annoying. Think about it.

To help you, PV provides two mechanisms. PV provides acces to his own memory management routines, they are *pvmalloc(), pvfree()* and *pvcalloc()*. If you want to allocate a memory block that may be used by PV, you should use these functions instead of the normal malloc(), free() or calloc() (the PV functions have exactly the same prototypes). In this way, you will have no problem with memory management runtime.
The second mechanism can help you to automate this task. If you define the PVMEM_OVERLOAD macro **before** including pvision.h in one of your files, then all *malloc()* in your code will be replaced by a call to *pvmalloc()* (same for th other functions). Be warned that you should not include other .h files after pvision.h (like stdlib.h) to prevent the macro from rewriting all malloc calls (do it only on YOUR files).

Notes about the software renderer

You will find in a material many field to customize how things are rendered, blended, filtered and so on ... The software renderer does not support all fields.
The software renderer apply the following rules:

- Textures are repeated not clamped
- Blending is not supported
- Depth buffering use CMP_LESS function (other are unimplemented)
- Alpha testing is CMP_ALWAYS
- Fog is ignored

Getting started with Panard Vision

The reader should look in the `samples` subdirectory where he will find simple tutorials on how to achieve some basic things with PV.
People that are interested in building file drivers or wants to mess with mesh can look in the `src` subdirectory where file drivers source lies.
People that want to use the tunable visibility pipeline should look in the `qkread` subdirectory where there is a sample to use the BSP tree and PVS infos contained in a quake level.
The sources of the Panard Viewer which is able to render Quake levels and 3Dstudio 4 files (with full animation support) is also provided.

API DESCRIPTION

Callback functions

Panard Vision uses many callback functions to interact with the host program. The addresses of the function to be called are stored in global variables.
Here is the list of the global call back variables and the associated prototypes for the user function:

PV_UserCleanUp *:*

Called when a fatal error occurred. Called by *PV_Fatal* before displaying anything. Application should stop every process that could crash the system like music player or timer critical routines.

Prototype: void PVAPI TheFunc(unsigned t)

Where t is the error code passed to *PV_Fatal*.

PV_UserLight :

Called when the lighting module needs to recompute the light with a type of PVL_USER_LIGHT.

Prototype: void PVAPI TheFunc(PVMesh *m,PVLight *l)

Where m is the mesh where the light should be recomputed and l the user light.

PV_UserMeshCleanUp :

Called just before a mesh is destroyed. Useful to free data stored in *UserData* by example.

Prototype: void PVAPI TheFunc(PVMesh *m)

Where m is the mesh being destroyed.

PV_UserCompileMaterials :

Called when a material with the USER_FX flag needs to be compiled. Useful to setup user rendering routines.

Prototype: int PVAPI TheFunc(PVMesh *o,PVMaterial *mat,unsigned FaceNumber)

Where o is the mesh concerned, material the source material compilation and FaceNumber the face number concerned in o. Returns an error code or cool if everything is ok.

PV_UserRenderWorld :

Called before all polygons of a frame are being sent to the renderer (hardware or software)

Prototype: int PVAPI TheFunc(PVWorld *,UPVD8 *)

Parameters are the same than those of *PV_RenderWorld*().

Global Variables

There is too many, I know, but some are useful.

RMaskSize,GMaskSize,BmaskSize
RedMask,GreenMask,BlueMask
RFieldPos,GFieldPos,BFieldPos

These are reflects of the parameters passed to *PV_SetRGBIndexingMode()*

PixelSize

Size of a RGB pixel in bytes

PV_Mode

Current Panard Vision working mode

PV_PipelineControl

Current Panard Vision working mode

Miscellaneous functions

void InitPVision(void)

MUST be called once before every other calls to PV functions, at the beginning of your program. Sets up internal variables.

int PV_SetMode(PVFLAGS mode)

Sets the working mode of Panard vision according to bit field in *mode*. *Mode* is a combination of the following flags :

PVM_PALETIZED8	PV will render in 256 colors palletized mode
PVM_RGB16	PV will render in 15/16 bits mode, shaded textures **MUST** be 256 colors palletized.
PVM_RGB	PV will render in true RGB mode (15/16/24/32 bit modes).

You must specify one of the three above.

PVM_SBUFFER	PV will use front to back drawing ordering, reducing overdraw to 0. Useful when overdraw is important.(ignored when PVM_USEHARDWARE is set)
PVM_MIPMAPPING	PV will perform MipMapping on materials with *TextureFlags* containing the *TEXTURE_MIPMAP* flag.
PVM_BILINEAR	PV will perform bilinear filtering on materials with *TextureFlags* containing the *TEXTURE_BILINEAR* flag.
PVM_TRILINEAR	PV will perform trilinear filtering on materials with *TextureFlags* containing the *TEXTURE_TRILINEAR* flag. (hardware only, if supported)
PVM_ALPHABLENDING	PV will perform alpha blending according to the materials settings. (hardware only, if supported)
PVM_ALPHATESTING	PV will perform Alpha-testing according to the materials settings. (hardware only, if supported)
PVM_USEHARDWARE	Enable hardware rendering. You mist have registered a hardware driver with *PV_SetHardwareDriver()*.
PVM_ZBUFFER	Enable Zbuffering on materials with ZBUFFER flag set.

You can combine one of the five above.

Example: *PV_SetMode(PVM_RGB16|PVM_SBUFFER|PVM_BILINEAR)* ;

NOTE: You cannot change the rendering state (*PVM_PALETIZED8, PVM_RGB16, PVM_RGB*) after the material compilation process ended. But you can change the other flags during run time.

PV_SetPipelineControl(PVFLAGS p)

Used to enable/disable part of the pipeline rendering. Combinable values for p are:

PVP_DISABLELIGHTING	Disable lighting computations, *user should fill ShadingInfo member of PVMesh*.
PVP_DISABLESORTING	Disable sorting stage, *Visible* member of PVWorld is not filled anymore, visible face are available in each *Visible* member of the PVMesh structs.
PVP_DISABLETRANSFORM	Disable transformation stage. *Rotated*, *Projected* members are not filled anymore.
PVP_NO_ZBUFFERCLEAR	PV will not perform Zbuffer clear on *PV_BeginFrame()*.
PVP_NO_TRIANGULATE	PV will not triangulate polygons before sending them to renderer.

Current state could be obtained via the global variable *PV_PipelineControl*

int PV_SetClipLimit(unsigned int minx,unsigned int maxx, unsigned int miny, unsigned int maxy, unsigned int ScanLine)

Sets the clipping window info for subsequent rendering.

Minx, maxx are integers standing for the x left and right coordinates of clipping window in the screen buffer.
Miny, maxy are integers standing for the y top and bottom coordinates of clipping window in the screen buffer.
Scanline is the size of a scanline in bytes in the screen buffer.

void PV_TriangulateFace(PVFace *f, void (PVAPI * Func)(PVFace *f))

Triangulate polygon described by *f* and call function *func* for each generated triangle. The original face (ie the PVFace structure before triangulation) is available in the *Poly* field of each PVFace generated (you will need a C-Typecasting to access this member). This gives a way to access the original n-gone.

void PV_GarbageCollector(PVWorld *w)

Recover memory allocated during render process. Should not be called too often because the buffer freed by this function may be needed by PV and hence take a little time to reallocate during the next rendering.

void PV_Fatal(char *s,unsigned int t)

Terminates program displaying the string *s* and the error code *t*. The callback function pointed by *PV_UserCleanUp* is called before displaying these info.

void *PV_GetFiller(PVFLAGS flags)

Returns a pointer to the filler routine that should be used to render a face with the rendering flags *flags*.

void __cdecl PV_Log(char *logfile,char *fmt, ...)

Log a formatted line *fmt* to the file *logfile*. This is a not buffered logging, after the call to *PV_Log()* the file is always updated.

void PV_BeginFrame(void)

This call must be called once before beginning rendering. It clears Z/Sbuffer (according to current pipeline control word).

void PV_EndFrame(void)

This call must be called once a rendering process is done. You can mix Mesh oriented API and Panard Primitives between *PV_BeginFrame()* and *PV_EndFrame()*.

void PV_PrepareFace(PVFace *f)

This function prepares a face for rendering (handles wrapping flags, texture coordinates generation and so on). It's published to give you the hability to hack some things with PV.

UPVD8 PV_LookClosestColor(PVRGB *pal,PVRGB c,unsigned reserved)

Look for the closest color of *c* in the 256 colors palette *pal* excluding the first *reserved* colors.
Returns index in *pal* of the closest color.

PVTexture *PVAPI PV_MipResample(UPVD8 *mip, unsigned width,unsigned height,char grow,PVFLAGS texflags,PVRGB *Pal,unsigned reserved)

This function resizes the texture pointed by *mip* to a power of 2 size. *Mip* is *width* pixels wide and *height* pixel tall. The resizing will be done with the next bigger power of 2 sizesif *grow* is set to 1 and to the next lower size otherwise. The type of texture being resampled is defined in texflags, which could be one of the following TEXTURE_PALETIZE8, TEXTURE_RGB or TEXTURE_RGBA (see material chapter). A palette for *mip* may be specified by *pal*. The number of colors that are reserved (for paletized textures) and that should not be used during the sampling process is specified in *reserved.*
The functions returns a PVTexture struct allocated on the stack (i.e. don't try to free it, make a copy of it after a call to the function) filed to point on the new texture. If the incoming texture is already a power of two the returned texture point on the incoming texture.

void *PVAPI pvmalloc(size_t size)

Wrapper to a PV compatible malloc. (see memory management chapter).

void *PVAPI pvcalloc(size_t num, size_t size)

Wrapper to a PV compatible calloc. (see memory management chapter).

void PVAPI pvfree(void *memblock)

Wrapper to a PV compatible free. (see memory management chapter).

Mathematical Functions

void SetupMatrix3x3(PVMat3x3 Matrix,float AngleX,float AngleY,float AngleZ)

Fill in *Matrix* according to the three angles of rotation (in radians) around axis x, y and z.

void RotatePoint3x3(PVPoint *v,PVMat3x3 m,PVPoint *p)

Multiply the *v* vector by the matrix *m*, store the result in the vector *p*.

void RotateInvertPoint(PVPoint *v,PVMat3x3 m,PVPoint *p)

Multiply the *v* vector by the invert matrix of *m*, store the result in the vector *p*.

void OrthonormalizeMatrix(PVMat3x3 m)

Orthonormalizes *m*.

void MatrixMul(PVMat3x3 m1,PVMat3x3 m2,PVMat3x3 m)

Multiplies matrix *m1* by *m2* and stores the result in *m*.

void QuatToMatrix(PVQuat *q,PVMat3x3 Matrix)

Generates the rotation matrix associated with the quaternion *q*. Result stored in the matrix *Matrix*.

void MatrixToQuat(PVMat3x3 Matrix,PVQuat *q)

Generates the quaternion associated to the rotation matrix *Matrix*. Result stored in the quaternion *q*.

void QuatExp(PVQuat *q, PVQuat *dest)

Returns in *dest* the exponential of *q*.

void QuatInv(PVQuat *q, PVQuat *dest)

Returns in *dest* the inverse of *q*.

void QuatLog(PVQuat *q, PVQuat *dest)

Returns in *dest* the logarithm of *q*.

void QuatLnDif(PVQuat *p, PVQuat *q, PVQuat *dest)

Returns in *dest* the logarithm of the relative rotation from *p* to *q*.

float QuatMod(PVQuat *q)

Returns the modulus of *q*.

float QuatDot(PVQuat *q1, PVQuat *q2)

Returns the dot product of *q1*by *q2*.

float QuatDotUnit(PVQuat *q1, PVQuat *q2)

Returns the dot product of *q1*by *q2*. (*q1* and *q2* should be normalized)

void QuatNegate(PVQuat *q)

Negates *q*.

void QuatMul(PVQuat *q1,PVQuat *q2,PVQuat *q3)

Multiply quaternion *q1* and *q2*, store result in *q3*.

void QuatToMatrix(PVQuat *q,PVMat3x3 Matrix)

Generates the rotation matrix associated with the quaternion *q*. Result stored in the matrix *Matrix*.

void MatrixToQuat(PVMat3x3 Matrix,PVQuat *q)

Generates the quaternion associated to the rotation matrix *Matrix*. Result stored in the quaternion *q.*

void QuatSlerp(PVQuat *q1,PVQuat *q2,PVQuat *dest,float time,float spin)

Performs a SLERP interpolation between quaternion *q1* and *q2* according to *time*. Result is stored in *qdest*. *time* represents the interpolation factor between 0 and 1. (t=0, q3=q1 ; t=1, q3=q2). *spin* is the number of PI radians to add to the rotation.

void QuatSlerpLong(PVQuat *q1,PVQuat *q2,PVQuat *dest,float time,float spin)

Same than QuatSlerp, but performs slerp along the longest arc.

void QuatScale(PVQuat *q,float s,PVQuat *dest)

Scale *q* by *s* and stores result in *dest.*

void QuatUnit(PVQuat *q,PVQuat *dest)

Unitize *q*, stores result in *dest*.

void OrientationToQuat(float x,float y,float z,float a,PVQuat *q)

Fills in the quaternion *q* according to a vector defined by *x,y,z* and an angle of rotation *a* (in radian) around this vector.

void QuatToOrientation(PVQuat *q, float *x,float *y,float *z,float *angle)

Fill in *x,y,z,angle* by the Orientation representation of *q* (direction+angle).

View Frustum handling

int PV_GetFrustumFlags(PVPoint *v)

This function classifies a 3D point with each planes of the frustum and each user defined planes in the active camera. This function is designed to be used in user visibility pipeline, see below and Quake driver.

int PV_ClipFaceToFrustum(PVFace *f)

This function clips a face to the current view frustum and user frustum. This function is designed to be used in user visibility pipeline, see below and Quake driver.

int PV_GetNbrFreeSpecialClipPlanes(void)

This function returns the number of planes available to the user (not the same than user defines clipplanes in cameras) for special effects (like portals).

PVPlane *PV_AllocClipPlanes(unsigned n)

This function allocates *n* clipplanes, and returns an array of PVPlane for accessing them. The number of free planes could be obtained by *PV_GetNbrFreeSpecialClipPlanes().*

void PV_FreeLastClipPlane(void)

Frees the last clip planes. I.e. if you previously allocated 3 clipplanes with *PV_AllocClipPlanes()*, after a call to this function you will be able to access only the first 2 of them.

void PV_FreeAllClipPlanes(void)

frees all allocated clip planes. Reset frustum system.

void PV_SetPlane(PVPlane *plane,double position,PVPoint *normal)

Sets values for a PVPlane, *position* is a point contained by the plane, and *normal* the normal to the plane.

unsigned PV_GetNbrClipPlanes(void)

Returns the actual number of clipplanes allocated by *PV_AllocClipPlanes().*

unsigned PV_SetNbrClipPlanes(unsignd nbr)

Sets the actual number of clipplanes.

MESH ORIENTED API

Introducing the world graph

Everything element of a world in the retained mode of PV is stored in the world graph. This graph is an n-ary tree. This gives the ability to perform hierarchical positioning, culling and some special techniques like portals.

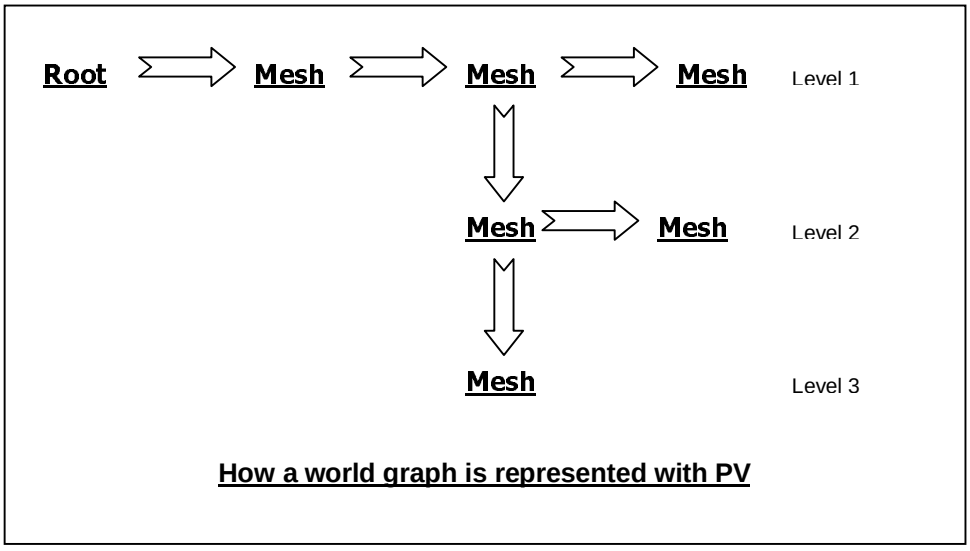
A mesh can have an unlimited number of children (children are added one by one with *PV_AddChildMesh()*). A child mesh can inherit position from its father (or not, this is chosen through the MESH_INHERIT flag, see PVMesh description). When a child-mesh inherits position from his father, then rotation and translation of the child-mesh is combined with those of his father before rendering.

When the rendering pipeline culls a father-mesh out (i.e.: mesh outside the view frustum for instance), all its children are still processed (i.e. tested through the visibility pipeline). But when the flag MESH_FORGET_CHILD is set on a father mesh, then, all his children are NOT processed anymore (even if the father child is visible). This allows for quickly "deactivates" part of a world to gain speed.

Hierarchical world modeling allows for hierarchical world culling as well. By tagging a mesh with the MESH_CULL_CHILD, then if the visibility pipeline culls this mesh all its children will be too. This way that is the father that affects the children behavior.
Another form of hierarchical culling is available, this time it's the children mesh that choose to inherit culling or not from his father with the MESH_INHERIT_CULL flag.
Hierarchical culling can be a great help for some indoor rendering, grouping all objects inside a room as children of the room mesh. This way when the room is not visible, all objects inside are simply forgot.

This hierarchical representation can be used to "group" mesh together, by defining an "empty" (i.e. with 0 faces) and adding children to it (with MESH_INHERIT flag enabled), you can manipulate a set of mesh only by manipulating their father.

The root node of a graph is accessible by the field *Objs* of a PVWorld object. There can not have cycle in the world graph, i.e. a mesh with a child already in the hierarchy (the only exception to this rule applies when using portal mesh, see portal section).



How a world graph is represented with PV

Cameras handling

Structure PVCam

This structure defines a camera for Panard Vision. You can create a cam with *PV_CreateCam*, and assign cam for rendering with *PV_AssignCam.*
You have two ways to control the camera position in space. You can use the *roll, pitch, yaw* method (*PV_SetCamAngles*) to define the cam position and target with angles of rotation around the cam basis (like in flight simulators by example). Alternatively, use the *target* field to set up the position according to what the camera should look at (useful to track a specific object by example).

Fields are:

- *Name* : Pointer to a zero terminated string containing the name of the camera.
- *fieldofview* : float value indicating the field of view for this cam. Good value is 1.6.
- *Height* : integer value representing the 'height' of the cam plane in pixels, habitually equal to the rendering window height.
- *Width* : integer value representing the 'width' of the cam plane in pixels, habitually equal to the rendering window width.
- *CenterX, CenterY* : These 2 fields indicates the coordinates of the center of the viewport, allowing for special effects.
- *roll, pitch, yaw* : float values standing for defining the position of the camera basis in world space.
- *pos* : position in world space of the camera.
- *target* : target of the camera in world space, i.e. what the camera is looking at.
- *Matrix* : World space to cam space transformation matrix. Direction of the cam is stored in *Matrix[2][x]*.
- *FrontDist* : Front clip plane distance (>0).
- *BackDist* : Back clip plane distance(>FrontDist).
- *StartingLocation* : Pointer to a PVMesh where the hierarchical rendering should start (NULL if all meshes should be processed). This is useful when portal rendering.
- *Flags* : Flags affecting the camera. Current valid flags are:

CAMERA_AUTOLOCATE	The camera *StartingLocation* member will be automatically filled by PV (the mesh pointed will be the one which contains the camera). This is used when using portal rendering.

PVCam *PV_CreateCam(char *n)

Creates a camera object named *n*. The contents of the zero terminated string pointed by n is copied in the field *Name* of the Cam structure. Returns a pointer to the *PVCam* object created.

NOTE: don't forget to set fields *Height, Width, CenterX, CenterY* and *fieldofview*, after creating the object.

void PV_RAZCam(PVCam *c)

Resets the values of the *c* PVCam object except for *Width,Height, fieldofview, Name*.

void PV_ComputeCam(PVCam *c)

Computes the *Matrix* field of the *c* PVCam object according to the content of *roll,pitch,yaw*. You will not normally have to call this directly.

void PV_CamAhead(PVCam *c,float a)

Move PVCam *c* in the direction of view by *a*. Used to move ahead relatively to the cam basis.

int PV_SetCamName(PVCam *c,char *n)

Sets the *Name* field of *c* to *n*. The zero terminated string pointed by *n* is copied to *Name*. Previous content is freed if needed.

void PV_SetCamFieldOfView(PVCam *c,float f)

Sets the *fieldofview* field of *c* to *f*.

void PV_SetCamAngles(PVCam *c,float yaw,float pitch,float roll)

Sets the fields *yaw, pitch, roll* of *c* to the arguments.

void PV_SetCamPos(PVCam *c,float x,float y, float z)

Sets the three members *xf,yf,zf* of the field *pos* of *c* to *x,y,z.*

void PV_SetCamTarget(PVCam *ca,float x,float y, float z)

Sets the three members *xf, yf, zf* of the field *target* of *c* to *x,y,z.*

void PV_SetCamRoll(PVCam *c,float roll)

Sets the *roll* angle (in radians) around view direction.

void PV_KillCam(PVCam *c)

Free the memory taken by *c*, *Name* string is freed if needed.

int PV_SetClipPlane(PVCam *c,unsigned i,PVPoint Pos,PVPoint Normal)

Sets values for the *i*th user clipplanes. *Pos* is a point included in the plane, *Normal* is the normal to the plane.

int PV_EnableClipPlane(PVCam *c,unsigned i,unsigned val)

Enables/Disables the *i*th user defined clipplane ([0..MAX_USER_CLIP_PLANES[).
Allowed values for *val* are :
PV_ENABLE : to enable the clip plane
PV_DISABLE : to disable the clip plane

PVMesh *PVAPI PV_FindCurrentCameraCell(PVMesh *m,PVCam *c)

Returns the mesh in which the camera *c* is currently inside. The camera *c* is classified on the hierarchy pointed by *m*.
This is useful when portal rendering to setup the starting point in the world-graph rendering.

Light handling

Structure PVLight

This structure defines a light in a world. After creating and setting up a light. You must add this light to a *PVWorld* object with *PV_AddLight.*
Fields are:

- *Name* : pointer to a zero terminated string containing the name of the light.
- *Type* : one of the following values : PVL_DIRECTIONAL, PVL_PARALLEL, PVL_INFINITEPOINT, PVL_POINT, PVL_SPOT, PVL_USER_LIGHT. Indicating the type of lighting performed by this light. (see below)
- *Flags* : one of the following flags : LIGHT_NORMAL (default), LIGHT_FORGET. When LIGHT_FORGET is set, the light is 'switched off'.
- *Color* : PVRGBF value indicating the color of the light (used only in PVM_RGB rendering mode, assumed to be white otherwise).
- *Intensity* : float value between 0 and 1 stating the power of the light.
- *Range* : Range of action for this light, used with PVL_SPOT and PVL_POINT lights.
- *Attenuation0 :* Constant light intensity. Specifies a light level that does not decrease between the light and the cutoff point specified by *range*. Valid for PVL_SPOT.
- *Attenuation1 :* Light intensity that decreases linearly. Valid for PVL_SPOT.
- *Attenuation2 :* Light intensity that decreases according to a quadratic factor. Valid for PVL_SPOT.
- *FallOff* : Decrease the illumination between the umbra (angle specified by *Theta*) and the penumbra (angle specified by *Phi*).
- *Position* : point indicating the position of the light in world space. Valid for PVL_PARALLEL, PVL_POINT, PVL_INFINITEPOINT,PVL_SPOT.
- *Direction* : vector indicating the direction of parallel light rays for PVL_DIRECTIONAL and PVL_PARALLEL.
- *Target* : position of the spot's target for PVL_SPOT lights.
- *UserData* : pointer to a user block of memory. Used to attach user specific info for this light.

Light types are:

PVL_DIRECTIONAL : Light casts parallel light rays using the *Direction* vector. Fastest lights to compute.
PVL_PARALLEL : Light casts parallel light rays using the *Direction* vector, on one side of the plane define by the *Position* point and the normal vector *Direction.* Casts parallel lights rays using the *–Direction* vector on the other side of this plane..
PVL_INFINITEPOINT : Light is a point at *Position* that casts non-parallel light rays in every directions.
PVL_POINT : Light is a point at *Position* that casts non-parallel light rays in every directions within *Range*.
PVL_SPOT : Light is a spot.
PVL_USER_LIGHT : Used to let the user computes the lighting in his specific way.

PVLight *PV_CreateLight(PVLightType f,char *name)

Allocates memory for a new light. Fill in the field *Type* with *f*, and the *Name* field with *name*. Returns a pointer the new light object.

int PV_SetLightName(PVLight *l,char *n)

Sets field *Name* of *l* to *n.* Frees previously allocated memory for *Name*.

void PV_KillLight(PVLight *l)

Frees the memory taken by *l*.

void PV_SetLightDirection(PVLight *l,float x,float y,float z)

Sets the three members *xf,yf,zf* of the field *Direction* of *l*. To *x,y,z.*

void PV_SetLightPosition(PVLight *l,float x,float y,float z)

Sets the three members *xf,yf,zf* of the field *Position* of *l*. To *x,y,z.*

void PV_SetLightTarget(PVLight *l,float x,float y,float z)

Sets the three members *xf,yf,zf* of the field *Target* of *l*. To *x,y,z.*

int PV_SetLightIntensity(PVLight *l,float x)

Sets the field *Intensity* of *l* to *x*.

void PV_SetLightFlags(PVLight *l,PVFLAGS x)

Sets the field *Flags* of *l* to *x*.

void PV_SetLightType(PVLight *l,PVLightType x)

Sets the field *Type* of *l* to *x.*

void PV_SetLightRange(PVLight *l,float x)

Sets the field *Range* of *l* to *x*.

void PV_SetLightAttenuation(PVLight *l,float a0,float a1,float a2)

Sets the fields *Attenuation0,Attenuation1,Attenuation2* of *l* to *a0,a1,a2*.

void PV_SetLightSpot(PVLight *l,float Theta,float Phi,float Falloff)

Sets the fields *Theta,Phi,FallOff* of *l* to *Theta,Phi,Falloff*.

Materials handling

Structure PVMaterialHint

Hints are stored in the following structure. They are used to driver the hardware driver for texture management. These are only hints and could be totally ignored by a given driver. The interpretations of the hints are totally driver dependent.

- *Quality* : Flags indicating the desired quality for textures. Possible values are :
  - *PHQ_NORMAL* default driver quality
  - *PHQ_HIGH* keeps as close as possible to the original image
  - *PHQ_LOW* uses less memory
- Location : Flags indicating the type of material. Possible values are :
  - *PHL_NORMAL* nothing to say
  - *PHL_STATIC* static material, will likely not change
  - *PHL_DYNAMIC* material is quite often modified

Structure PVMaterial

This defines a material in a world. A material combines rendering information and all textures information in one object.
Fields are

- *Name* : name of the material, used to correlate with the material name from the face structure.
- *Type* : flags stating the rendering type for this material. (See below).
- *NbrMipMaps* : Number of desired mipmaps for mipmapped textures.
- *TextureFlags* : Flags describing the texture. (See below).
- *RepeatU, RepeatV* : These 2 fields determine how texture will be treated (for U and V axis) when texture coordinates are not in [0..1], the values allowed are TEXTURE_WRAP (the texture is repeated across the polygon) and TEXTURE_CLAMP (the coordinates are clamped to [0..1]). Software rasterizer only supports TEXTURE_WRAP.
- *DepthTest* : Kind of test performed to accept or reject pixels when zbuffering, allowed values are:
  - *CMP_LESS* : passes when incoming depth is less than stored z.
  - *CMP_NEVER :* passes never.
  - *CMP_EQUAL :* passes when incoming and stored values are equal.
  - *CMP_LEQUAL:* passes when incoming depth is less or equal than stored depth.
  - *CMP_GREATER:* passes when incoming depth is greater than stored depth.
  - *CMP_NOTEQUAL:* passes when incoming value is different of the stored depth.
  - *CMP_GEQUAL:* passes when incoming value is greater or equal than stored depth.
  - *CMP_ALWAYS:* passes always.

  Note: Software rasterizer only supports CMP_LESS.
- *AlphaTest* : Kind of test performed to accept or reject pixels when alphatesting, allowed values are the same than for *DephtTest*. The comparisons are done between the alpha component of each pixel and the Alpha reference value (see below). (ignored in software)
- *AlphaReference* : Values used for comparison when Alpha Testing. (ignored in software)
- *BlendRgbSrcFactor* : Source factor used for RGB component when alpha blending. (ignored in software)
- *BlendRgbDstFactor* : Destination factor used for RGB component when alpha blending. (ignored in software)
- *BlendAlphaSrctFactor* : Source factor used for Alpha component when alpha blending. (ignored in software)
- *BlendAlphaDstFactor* : Destination factor used for Alpha component when alpha blending. (ignored in software)
- *AlphaConstant* : Constant alpha value for a the material (ignored for the moment).
- *Pal* : Pointer to a 256 *RGB* array storing the palette of the texture if needed.

- *StartingColorIndex* : entry index in the global palette of the world where this material is attached to for interpolating shading in FLAT|NOTHING and GOURAUD|NOTHING in PVM_PALETIZED8 mode.
- *EndingColorIndex* : entry index in the global palette of the world where this material is attached to for interpolating shading in FLAT|NOTHING and GOURAUD|NOTHING in PVM_PALETIZED8 mode. The shading will be interpolated between the two colors referenced by *StartingColorIndex* and *EndingColorIndex*.
- *Diffuse :* PVRGBF Value indicating the diffuse value for this material (color of the material in GOURAUD and FLAT shading mode).
- *Specular :* PVRGBF Value indicating the specular color for this material.
- *Emissive :* PVRGBF Value indicating the emissive color for this material.
- *SpecularPower :* integer indicating the sharpening of the *specular* artifact.
- *UserData* : 32 bits left to the user to store special info for this material (example a pointer to another struct).
- *Hint* : Hints for texture usage (see Material Hints below).
- *PaletteTranscodeTable* : is a pointer to a 256*256*1 bytes array holding color translation table for lighted and mapped materials in PVM_PALETIZED8 modes. This table gives the final colors depending on the incoming non shaded texture color index, and a light intensity. This table is automatically computed by PV or can be setup by user. To deactivates automatic PV lighttable computation and to specify your own, set this member to your own array before calling *PV_CompileMaterials* or *PV_CompileMaterial*.
- *RGB16TranscodeTable* : same than *PaletteTranscodeTable* but for PVM_RGB16 modes, this is a 256*256*2 bytes array, holding 16 bits values (colors that will be plotted to screen).

NOTE: Specular is only supported in non PVM_RGB modes.

The *Type* field is a combination of rendering flags, shading flags and modifier flags. These flags are :

Shading Flags :

NOTHING	No shading
FLAT	Flat Shading
GOURAUD	Gouraud Shading
PHONG	Fake Phong Shading (lights are not taken into accounts, the virtual light is at the camera place)
BUMP	Fake Bump Mapping, add height to mapping (lights are not taken into accounts, the virtual light is at the camera place)
U_PHONG	like PHONG but auxiliary texture is supplied by user.
U_BUMP	like BUMP but auxiliary texture is supplied by user.
LIGHTMAP	Use lightmap rendering, each face with a material setting this flag should have Lightmap attached.

Rendering Flags :

NOTHING	No special rendering, pure shading
MAPPING	Affine texture mapping
AMBIENT_MAPPING	Affine texture mapping with generated mapped coordinates according to cam pos around faces

See Annex for a list of currently supported combination

Modifier Flags :

ZBUFFER	Faces will be Zbuffered if PVM_ZBUFFER is set *Note : You should define all the materials with the ZBUFFER flag to obtain a coherent display. But it could save time to define this on, by example, two particular materials that are known to cause trouble at display.*
PERSPECTIVE	Add perspective correction
AUTOMATIC_BILINEAR	Add automatic bilinear switch on/off based on mipmap level of a face
AUTOMATIC_PERSPECTIVE	Add automatic perspective switch on/off based on distance of a face from camera
USER_FX	Used to defines a user type material.

DOUBLE_SIDED	The faces with this material will not be backface culled. Thus, faces will be visible from both sides.
COLORTABLE_DONT_FREE	The color tables (pointed by *PaletteTranscodeTable* and *RGB16TranscodeTable*) will not be freed when a material is destroyed. Useful for user provided tables.

The *TextureFlags* field is a combination of texture type flags and texture special features flags :

Texture type flags :

TEXTURE_NONE	The material have no texture
TEXTURE_PALETIZED8	The supplied texture is a palletized 256 colors textures. Palette is pointed by *Pal*.
TEXTURE_RGB	The supplied texture is a 24 bits RGB texture pointed by *Tex* stored as Red,Green,Blue.
TEXTURE_RGBA	The supplied texture is a 32 bits RGBA texture pointed by *Tex* stored as Red,Green,Blue,Alpha. **WARNING:** This flag is not currently used in PV, it's only here to permit use of alpha channel in user routines, **DON'T USE IT WITH BUILTIN PV ROUTINES, or you will get unpredictable results.**

Note : If you are in PVM_RGB16 mode you must use palletized textures on shaded materials !

Special features flags :

TEXTURE_MIPMAP	Texture will be MipMapped if PVM_MIPMAP is set with *PV_SetMode*.
TEXTURE_BILINEAR	Texture will be bilinear filtered if PVM_BILINEAR is set with *PV_SetMode*. (only in PVM_RGB mode).
TEXTURE_TRILINEAR	Texture will be trilinear filtered. (hardware only)
TEXTURE_MANUALMIPMAPS	The mipmaps for this texture are computed by host program, PV will not compute them. You should set mipmaps with *PV_SetMaterialMipMap*().
TEXTURE_TEX_DONT_FREE	The PV runtime should not try to deallocate himself the Textures and Palettes when killing a material.
TEXTURE_ATEX_DONT_FREE	The PV runtime should not try to deallocate himself the AmbientTexture when killing a material.

PVMaterial *PV_CreateMaterial(char *n,PVFLAGS type,PVFLAGS flags,unsigned char nbrmipmaps)

Creates a new material object with the name *n*, the type *type*, the TextureFlags *flags*, and the desired number of MipMaps sets to *nbrmipmaps*.

void PV_KillMaterial(PVMaterial *m)

Destroy PVMaterial pointed by *m*. Frees all memory pointed by *Tex*, *Pal* and so on….

void PV_SetMaterialTextureFlags(PVMaterial *m,PVFLAGS f)

Sets the *TextureFlags* field of *m* to *f*.

void PV_SetMaterialType(PVMaterial *m,PVFLAGS f);

Sets the *Type* field of *m* to *f*.

int PV_SetMaterialTexture(PVMaterial *m,unsigned width,unsigned height,UPVD8 *texture,PVRGB *pal)

Assign to a mapped material *m* a texture pointed by *texture* with a size of *width*height* pixels. If the texture is palletized a 256 entry RGB array representing the palette must be passed with *pal*.

NOTE: The size of the texture **MUST** be a power of 2. Example 64*32 is valid, 68*128 is not. The storing method of the texture must correspond to the *TextureFlags* field of the material.

int PV_SetMaterialAuxiliaryTexture(PVMaterial *m,unsigned width,unsigned height,UPVD8 *texture)

Sets the info for the auxiliary texture use in U_BUMP and U_PHONG rendering mode. *Texture* is a pointer to a 8 bit texture (see this as a « light » texture).

NOTE: The size of the texture **MUST** be a power of 2. Example 64*32 is valid, 68*128 is not. The auxiliary texture must be the same size than the primary texture when the auxiliary texture is a BumpMap.

void PV_SetMaterialPureColorsIndex(PVMaterial *m,UPVD8 start,UPVD8 end)

Sets the fields *StartingColorIndex* and *EndingColorIndex* to *start* and *end*.

int PV_SetMaterialLightInfo(PVMaterial *m,PVRGBF Emissive,PVRGBF Diffuse,PVRGBF Speculer,unsigned specularpower)

Sets the fields *Emissive, Diffuse, Specular* and *SpecularPower* according to arguments.

int PV_SetMaterialColor(PVMaterial *m,float r,float g,float b)

Sets the three members *xf,yf,zf* of the field *Color* of *m*. To *x,y,z.*

int PV_BuildBumpMap(PVMaterial *m,UPVD8 *Texture,unsigned width,unsigned height,float scale)

Creates a Bump Mapping (for BUMP and U_BUMP rendering) based on a height field pointed by *Texture.* The height field is *width*height* bytes large. The *scale* factor will be applied to the generated bump map.

int PV_SetRGBIndexingMode(UPVD8 RMaskSize,UPVD8 GMaskSize,UPVD8 BMaskSize, UPVD8 RFieldP,UPVD8 GFieldP, UPVD8 BFieldP,UPVD8 AMaskS)

Sets the desired output pixel format for PVM_RGB and PVM_RGB16 mode.
RmaskSize is the size of the Red component mask. (8 for 24 bits mode).
GmaskSize is the size of the Green component mask. (8 for 24 bits mode).
BmaskSize is the size of the Blue component mask. (8 for 24 bits mode).
RFieldP is the position of the Red component. (16 for 24 bits mode).
GFieldP is the position of the Green component. (8 for 24 bits mode).
BFieldP is the position of the Blue component. (0 for 24 bits mode).
AmaskS is the size of the reserved field (sometimes called alpha component), 0 in 24 bits modes, 8 in 32 bits modes

PVD32 PV_MakeNormalizedColorRGB(PVRGB c)

This function converts the *r, g, b* values of *c* into a 32bits value which can be directly sent to the rendering windows (according to values set by *PV_SetRGBIndexingMode()*), and returns it.

void PV_SwapMaterials(PVMaterial *m1,PVMaterial *m2)

Swaps *m1* and *m2*. This could be used to perform animated texture mapping.
For instance, suppose you have an *n* framed texture. Set a material as the root texture with the name 'MyTex 0', and load all the other frames with *PV_AddMaterial ('MyTex 1', 'MyTex 2' ...).* Sets face that should be animated with the 'MyTex 0' materials, and then at run time swap 'MyTex 0' with 'MyTex 1', after 'MyTex 0' with 'MyTex 2' and so on.

int PV_MaterialGammaCorrect(PVMaterial *m,float gamma)

Gamma corrects textures of *m* according to *gamma.*

int PV_GammaCorrectPal(PVRGB Pal[256],float gamma)

Gamma corrects a palette *Pal* by *gamma*.

int PV_SetMaterialMipMap(PVMaterial *m,unsigned mipnbr,UPVD8 *texture)

Sets the mipmaplevel *mipnbr* of the main texture in *m* to *texture*. *Mipnbr* should be between 1 and *m->NbrMipMaps*-1. To set the main texture use *PV_SetMaterialTexture()*. Each mipmap level should be half the size in width and height than the precedent.

PVTexBasis *PV_CreateTextureBasis(float s[4],float t[4])

Creates a basis for an orthogonal projection texture mapping coordinates generation. The 3 first coords are the unitary base vector for the s and t axis, the last coord is the origin in the texture.

void PV_KillTextureBasis(PVTexBasis *b)

Kill a basis allocated by *PV_CreateTextureBasis()*.

void PV_AttachTextureBasis(PVFace *f,PVTexBasis *b)

Attach texture basis *b* (created by *PV_CreateTextureBasis()*) to the face *f*. The TEXTURE_BASIS flag is not added to *f* by this function. You have to do it as well.

Notes about alpha blending

The parameters for alpha blending correspond to the parameters of this formula :

FinalColor= RGBSourceFactor*SourceColor+RGBDestinationFactor*DestinationColor

Where SourceColor is the color of the incoming pixel and DestinationColor is the color of the corresponding frame buffer pixel.
This formula is applied to the r, g and b component of each color and, if the hardware does not support separate factors for alpha, to the alpha channel. If the hardware supports different factors for alpha channel then the same formula is used with AlphaSourceFactor and AlphaDestFactor.
The Values for the factors are:

BLEND_ZERO	=0
BLEND_ONE	=1
BLEND_DST_COLOR	=The destination color
BLEND_ONE_MINUS_DST_COLOR	=1-The destination color
BLEND_SRC_ALPHA	=Alpha of the incoming pixel
BLEND_ONE_MINUS_SRC_ALPHA	=1-Alpha of the incoming pixel
BLEND_DST_ALPHA	=Alpha of the destination pixel (needs support for alpha buffer).
BLEND_ONE_MINUS_DST_ALPHA	=1- Alpha of the destination pixel (needs support for alpha buffer).
BLEND_ALPHA_SATURATE	=min(source alpha,1-destination alpha) (needs support for alpha buffer). This factor is only valid for source factors.

Meshes handling

Structure PVMesh

This structure defines a mesh in a world. A mesh is an ensemble of vertex, faces and info related to them. A mesh has a position in a PVWorld .
Fields are:

- *Name* : string containing the name of the mesh.
- *Flags* : Flags affecting the mesh, possible combinable flags are:
  - *MESH_FORGET* hide this mesh in a world.
  - *MESH_FORGETCHILD* hide children of this mesh.
  - *MESH_FORGETALL* combination of MESH_FORGET and MESH_FORGETCHILD. The mesh and his children are not processed.
  - *MESH_INHERIT* the position and rotation of the mesh are relative to his father position and rotation (hierarchical positioning).
  - *MESH_PORTAL* This mesh is a portal. (Sets by PV)
  - *MESH_NOSORT* bypass sorting stage for this mesh (see Quake reader sample).
  - *MESH_NOLIGHTING* bypass lighting stage for this mesh.
  - *MESH_CULL_CHILD* If this mesh is culled during the frustum clipping, then all its children will be too.
  - *MESH_INHERIT_CULL* This mesh will be culled if his father is.
- *NbrFaces* : Number of faces (triangles) contained by this mesh.
- *NbrVertices* : Number of vertices contained by this mesh.
- *NbrVisibles* : Number of visible faces during last compute process.
- *Pivot* : Position of the rotation center of the object in world space. (usually gravity center).
- *ScaleX* : Scaling factor for X-axis.
- *ScaleY* : Scaling factor for Y-axis.
- *ScaleZ* : Scaling factor for Z-axis.
- *UserData* : Pointer left to the user.
- *Owner* : Pointer to the world that owns this mesh.
- *NbrSwitchItems* : in case of a switchable node, this member indicates the number of switch path available.
- *UserPipeline* : if this pointer is set then the default visibility pipeline of PV is bypassed and the provided function is called instead. The prototype for the function is:
  *int (PVAPI * UserPipeline)(struct _PVMesh *m)*
  The return value of this function will indicates PV what to do with clipping and lighting (See the Quake sample, and PVISION.H for possible return values).

PVMesh *PV_CreateMesh() (DEPRECATED)

FUNCTION DEPRECATED, USE PV_SIMPLECREATEMESH() INSTEAD.

PVMesh *PV_SimpleCreateMesh(unsigned nbrfaces,unsigned nbrvertices,unsigned maxclipface,unsigned maxclipvertex)

Creates a Mesh Object with *nbrfaces* face *nbrvertices* vertices and room for clipping up to *maxclipface* faces and *maxclipvertex* vertices, returns a pointer to the new PVMesh structure.
All buffers are allocated (Mapping, Shading, Vertex, Face, Visible, Rotated, Projected) and can be used directly.

void PV_KillMesh(PVMesh *no)

Free memory used by the PVMesh *no*.

void PV_MeshTranslateVertices(PVMesh *o,float x,float y,float z)

Adds the vector formed by *x*, *y*, *z* to each vertex contained by *o*. This is slow, and affects really the mesh. To move a mesh in a world it is preferable to directly set the *Pos* member of *o*. This function destroys the associated collision tree if there is one. It must be rebuilt.

void PV_MeshRotateVertices(PVMesh *o,PVMat3x3 m)

Rotates each vertices of *o* by *m*. the pivot point is assumed to be 0,0,0. This is slow, and affects really the mesh. To rotate a mesh in a world it is preferable to directly set the *Matrix* member of *o*. This function destroys the associated collision tree if there is one. It must be rebuilt.

void PV_MeshScaleVertices(PVMesh *o,float a,float b, float c)

Scale each vertices with *a* for x-axis, *b* for y-axis, *c* for z-axis. This is slow, and affects really the mesh. To scale a mesh in a world it is preferable to directly set the *ScaleX,ScaleY, ScaleZ* members of *o*. This function destroys the associated collision tree if there is one. It must be rebuilt.

int PV_MeshNormCalc(PVMesh *o)

Computes normal vectors for all faces of *o*.

void PV_MeshBuildBoxes(PVMesh *o)

Computes bounding boxes for *o*.

void PV_MeshSetupMatrix(PVMesh *o,float AngleX,float AngleY,float AngleZ)

Sets the rotation matrix of *o* according to the desired rotation angles (in radians) around x,y and z-axis.

void PV_MeshSetupMatrixDirect(PVMesh *o,PVMat3x3 mat)

Sets the rotation matrix of *o* to *mat.*

void PV_MeshSetupInvertMatrixDirect(PVMesh *no,PVMat3x3 t2)

Sets the rotation matrix of *o* to the invert of *mat*.

void PV_MeshSetupPos(PVMesh *o,float X,float Y,float Z)

Sets the translation vector of *o* (ie the position of the mesh center in world coordinates) to *(x,y,z).*

void PV_MeshSetFlags(PVMesh *o,PVFLAGS f)

Sets the *Flags* member of *o* to *f*.

PVMesh *PV_CreateMeshPortal(void)

Creates a mesh that will act as a portal in the world.

void PV_SetupPortal(PVMesh *m)

Call this on a portal mesh after seeted up his polygons info to make the protal usable.

PVMesh * PV_CloneMesh(PVMesh *m)

Returns a clone of *m*. The returned mesh is a full copy of *m.*

int PV_GetMeshAAExtent(PVMesh *m,PVPoint *org,PVPoint *extent)

Returns in the *PVPoint* pointed by *org* and *extent*, the origin and extent along the 3 axis of the axis aligned bounding box of *m*.

void PV_TransformPointLikeMesh(PVPoint *src,PVMesh *m,PVCam *c,PVPoint *rotated,PVScreenPoint *dest)

Transforms *src* like *m* seen through camera *c*. The final world coordinates vertex is stored in the *PVPoint* pointed by *rotated*. If *dest* is not NULL, the projected info for the vertex will be stored in the *PVScreenPoint* pointed by *dest*.

Switchable nodes

Switchable nodes allow you to select one path of the world graph dynamically at run time. This can be used to render LODs of a mesh.
The switch is done by a callback provided by client application, the return value of the callback is the number of the path to be rendered.

PVMesh * PV_CreateSwitchNode(PVMesh * (PVAPI *callback)(PVMesh*))

Creates a switch node. Each time the node will be processed by the render pipeline of PV, *callback* will be called with the node in parameter.

int PV_AddSwitch(PVMesh *switchnode,PVMesh *switchitem)

Adds a switch path to a switchable node. *Switchitem* is added to *switchnode*. If return value is nefative, then an error occured (negates the returned value to get a PV error code). Otherwise returns the index number of the added switch path (usable as a return value for callbacks).

PVMesh * PV_RemoveSwitch(PVMesh *switchnode,int itemnbr)

Removes from the switch node *switchnode*, the switchpath number *itemnbr*. Returns the switch path.

PVMesh * PV_GetSwitchItem(PVMesh *switchnode,int itemnbr)

Gets the switch path *itemnbr* from *switchnode*.

Mirrors handling

Structure PVMirror

This structure defines a planar geometrical non-recursive mirrors. The important fields are :

- *Flags* : Modifier for the mirror, allowed combinable values are :
  PV_MIRROR_FORGET will ignore this mirror
- *NbrVertices* : Number of vertices for this mirror, up to 8 vertices may be specified (minimum 3).
- *Vertex* : array of *NbrVertices PVPoint* that specify the coordinates of the vertex of the mirror.
- *Father* : pointer to a PVMesh object. If set the mirror will mime the PVMesh movements.
- *Matrix, Pivot, Pos* : Same as for a PVMesh object.
- *CutDist* : Maximum distance where the mirror will reflect (>0).

PVMirror * PV_CreateMirror(void)

Creates a mirror object.

void PV_KillMirror(PVMirror *m)

Kill a mirror object allocated by *PV_CreateMirror()*.

void PV_SetupMirror(PVMirror *m)

This function prepares a mirror for use, should be called once after the *Vertex* member of a mirror is modified.

World handling

Structure PVWorld

This structure defines a world. A world is a collection of mesh, materials and lights and describe what will be rendered.
Fields are :

- *NbrFaces* : Number of faces in the world. This is the sum of all the faces of all the meshes contained in the world.
- *NbrVertices* : Number of vertices in the world. This is the sum of all the vertices of all the meshes contained in the world.
- *NbrVisibles* : Number of faces visible in the world during the last compute process. This is the sum of all the visible faces of all the meshes contained in the world.
- *AmbientLight* : PVRGBF value indicating the color of the ambient light.
- *Global256Palette* : Used when in PVM_PALETIZED8 mode. Holds the resulting palette from the color quantization process of all the textures contained in the world. This pointer to a 256 array of RGB records.
- *ReservedColors :* Used when in PVM_PALETIZED8 mode. Indicates the number of colors to reserve in the global 256 colors palette resulting of the quantization of all the materials of the world. The first *ReservedColors* of the *Global256Palette* array are left to the user.
- *Camera :* Pointer to a PVCam struct. This will be the camera used to render this world. **DON'T FORGET TO INITIALIZE THIS MEMBER BEFORE RENDERING !**
- *Fog:* PVFogInfo describing fog infos for the world. All mesh rendered by this world will be fogged.

PVWorld *PV_CreateWorld(void)

Allocates memory for a PVWorld object, returns pointer to the newly created object.

void PV_KillWorld(PVWorld *w)

Free up memory used by the PVWorld object *w.*

void PV_RenderWorld(PVWorld *w,UPVD8 *Scr)

Renders the actual image computed by *PV_ComputeWorld* to the buffer pointed by *Scr* (may be directly the screen). According to the actual *PV_Mode* and Rgb indexing mode. When in hardware mode, the *Scr* parameter indicates the number if the surface to be rendered to :

0 always means BackBuffer (if there is one, or Front Buffer otherwise), this is a good value ☺
1 means FrontBuffer
2 means BackBuffer
and so on.

void PV_AddMaterial(PVWorld *w,PVMaterial *m)

Adds a material *m* created by *PV_CreateMaterial* to the world *w.*

PVMaterial *PV_GetMaterialPtrByName(char *name,PVWorld *z)

Retrieves a pointer to the material named *name* in the world *w.*

void PV_AddMesh(PVWorld *w,PVMesh *no)

Adds a mesh *no* created by *PV_CreateMesh* to the world *w.*

int PV_RemoveMeshByPtr(PVMesh *m)

Removes and destroy mesh *m* from his world.

int PV_RemoveMeshByName(PVWorld *w,char *name)

Removes and destroy mesh named *name* from the world *w.*

PVMesh *PV_GetMeshPtr(PVWorld *w,char *s)

Retrieves a pointer to mesh named *s* in the world *w.*

int PVAPI PV_AddChildMesh(PVMesh *father,PVMesh *child)

Add a child mesh *child* to the mesh *father*. *Child* MUST be a single mesh not already ina hierarchy and with no children.

PVMesh *PVAPI PV_UnlinkMesh(PVMesh *o)

UnLink the mesh *o* from his hierarchy. The mesh returned can be added to another world.

int PV_WorldSetAmbientLight(PVWorld *w,PVRGBF a)

Sets the *AmbientLight* field of the world *w* to *a*.

void PV_AddLight(PVWorld *w,PVLight *l)

Adds the light *l* created by *PV_CreateLight* to *w.*

int PV_RemoveLightByName(PVWorld *w,char *s)

Removes and destroy the light named *s* from *w.*

int PV_RemoveLightByPtr(PVWorld *w,PVLight *o)

Removes and destroy light *o* from *w.*

PVLight *PV_GetLightPtr(PVWorld *w,char *s)

Retrieves pointer on the light named *s* in the world *w.*

void PV_ScaleWorld(PVWorld *w,float ax,float ay,float az)

Scales all meshes of the world *w* by *ax,ay,az.* This function calls *PV_MeshScaleVertices*.

int PV_CompileMeshes(PVWorld *z)

This routine must be called once before processing a world. It prepares mesh for lighting calculations according to materials of the world. After calling this function you cannot change the material types of the world (but textures could be).

int PV_CompileMesh(PVMesh *m)

This routine is used to recompile just the mesh *m* in his world. You MUST have called *PV_CompileMeshes()* once on the world before.

int PV_CompileMaterials(PVWorld *w,UPVD8 (*CalcFunc)(PVRGB *pal,PVMaterial *m,unsigned col,unsigned lightlevel,PVRGBF ambient,unsigned reserved),UPVD16 (*CalcFunc16)(PVMaterial *m,unsigned col,unsigned lightlevel,PVRGBF ambient))

This function is used to convert material textures and other things to a more real-time aware fashion. By example RGB textures may be converted to something which will be faster to draw. After a call to this routine your textures are frozen. The last 2 parameters are used in PVM_PALETIZED8 mode or PVM_RGB16 mode. They are callback function used to compute translation table (color+shading=color). If none are provided (i.e. NULL value), PV use it's built-in lighting function.
If you want to use fun lighting table this function is for you (and *PV_LookClosestColor()* too).

Prototype for the function used in PVM_PALETIZED8 mode is:

UPVD8 ZeFunc(PVRGB *pal,PVMaterial *m,unsigned col, unsigned lightlevel, PVRGBF ambient,unsigned reserved)

Where:
pal is the global 256 colors palette where you have to choose the final color.
M is a pointer to the material being compiled.
Col is the number in pal of the color to be lighted
Lightlevel is the level of light from 0 to 255
Ambient is the RGB value describing the ambient light color in this world
Reserved the number of reserved colors in pal

The function returns the index number in *pal* of the selected color corresponding to *col* shaded with an intensity of *lightlevel*.

Prototype for the function used in PVM_RGB16 mode is:

UPVD16 ZeFunc(PVMaterial *m,unsigned col, unsigned lightlevel, PVRGBF ambient)

Where :
M is a pointer to the material being compiled.
Col is the number in pal of the color to be lighted
Lightlevel is the level of light from 0 to 255
Ambient the RGF value stating the ambient light color in this world

The function returns the RGB color corresponding to the color indexed by *col* in the *mat*'s texture palette, shaded with an intensity of *lightlevel*.
Hint: *PV_MakeRGBNormalizedColor()* is for you.

void PV_SetSortOrder(PVWorld *w,PVFLAGS f)

Selects the sorting order for the world *w*. f could be one of the following (not combinable) :

PVW_BACK_2_FRONT	Performs back to front sorting
PVW_FRONT_2_BACK	Performs front to back sorting
PVW_MATERIAL	Performs per material sorting, useful to speed up hardware rendering.

int PV_CompileMaterial(PVMaterial *m,UPVD8 (*CalcFunc)(PVRGB *pal,PVMaterial *m,unsigned col,unsigned lightlevel,PVRGBF ambient,unsigned reserved),UPVD16 (*CalcFunc16)(PVMaterial *m,unsigned col,unsigned lightlevel,PVRGBF ambient),PVRGBF ambient,unsigned reservedcolors)

Same as *PV_CompileMaterials()* but acts only on ONE material instead of all materials of a PVWorld. The material being compiled is specified by *m*. This function is very useful when using Panard Primitives.

void PV_AddMirror(PVWorld *w,PVMirror *l)

Adds the mirror *l* created by *PV_CreateMirror* to *w*.

int PV_RemoveMirrorByPtr(PVWorld *w,PVMirror *o)

Removes and destroy mirror *o* from *w*.

Collision detection

Structure PVCollidingFaces

This structure holds pair of colliding faces computed by *PV_MeshesColliding()*.

- *Face1* : pointer to a face of the first mesh.
- *Face2* : pointer to a face of the second mesh.

int PV_MeshComputeCollisionTree(PVMesh *m)

Compute collision info for *m*. This pass is required before calling other collision detection primitive.
Warning: This process can be very long.

int PV_CollisionTest(PVMesh *m1,PVMesh *m2,PVFLAGS flags)

Performs a collision test beetween *m1* and *m2*. *flags* indicates how collision computation will be performed. Valid values are :

COLLISION_NORMAL	The more precise type (the slowest too). Collision are accurate, and a collision table is filled with a list of faces that are colliding beetween the 2 mesh. These infos can be retrieved with *PV_GetCollisionTable()* and *PV_GetNbrCollision()*.
COLLISION_NO_REPORT	The function returns either COLLIDE_YES or COLLIDE_NO if *m1* and *m2* are colliding or not. But no informations about faces that are colliding are available (faster).
COLLISION_FAST	Use a faster collision test, but less precise (maybe wrong in some very complex environnement). No infos about faces are availlable.

PVCollidingFaces *PV_GetCollisionTable(void)

Return a pointer to the collision table generated by *PV_CollisionTest ()*. This table will be destroyed with the next call to *PV_CollisionTest()*.

unsigned PV_GetNbrCollision(void)

Returns the number of records in the collision table.

void PV_MeshKillCollisionTree(Pvmesh *m)

Frees memory used by the collision info for *m*.

PVMesh *PV_CreateDummyMesh(float radiusx,float radiusy,float radiusz)

Returns a pointer to dummy, invisible, mesh, which could be used to simulate camera object in collision detection. *Radius* is the size of the object along each axis.
Returns NULL if an error occurred.
The returned mesh has a collision tree already attached to it.

PANARD PRIMITIVES

Panard Primitives

This set of API calls enable direct drawing of polygons either in 2D or 3D, this is the same philosophy than OpenGL glBegin(), glVertex(), glEnd() system.
Panard Primitives use a state model, i.e. states are set by simple calls and remain in effects until reseted. All calls begin with « pv ». Calls do not return error codes, instead they set an internal error code which can be retrieved by *pvGetErrorCode()*.
For simplicity, PP uses the same macro-objects than the higher level of PV (i.e. PVCamera, PVLight and PVMaterial are used the same way).
The *PV_Mode* and *PV_PipeLineMode* are used by the primitive subsystem in the same way than the Mesh subsystem.

The primitives subsystem supports 2 working modes, 2D and 3D.In 2D, Panard Vision acts as a 2D renderer where you can throw primitives with direct screen coordinates (perspective correction and depth buffering may be used thus).
In 3D, vertices are specified in 3D space and are transformed/projected by the primitive subsystem.

The Panard Primitives subsystem is not a replacement to the mesh oriented function of PV. If you can « meshize » a set of vertex/faces do it and use the mesh possibility of PV! ! Performances will be better.

A common usage of the system will be:

```
... Some setup code...

PV_BeginFrame()

pvSetMaterial(...)
pvBegin(PV_TRIANGLES_STRIP,Screen)
pvColor(..)
pvMappingCoord(...)
pvVertex(...)

pvColor(..)
pvMappingCoord(...)
pvVertex(...)

pvColor(..)
pvMappingCoord(...)
pvVertex(...)

pvColor(..)
pvMappingCoord(...)
pvVertex(...)

pvEnd()

PV_EndFrame()
```

That would have drawn 2 triangles.

void pvReset(void)

This call resets the primitive subsystem. Subsystem is set to PV_2D, PV_NOLIGHTING, PV_CULL_NONE all lights are deleted.

void pvSetMode(pvMODE mode)

Sets if forthcoming vertices should be interpreted as 2D or 3D information. Valid value for *mode* are :
PV_2D for 2D drawing
PV_3D for 3D drawing

int pvGetErrorCode(void)

Return the error code of the last operation. See error-handling chapter.

void pvSetCamera(PVCam *cam)

Sets the camera object used when rendering in PV_3D mode.

void pvSetModelMatrix(PVMat3x3 mat)

Sets the rotation matrix applied to 3D vertex when in PV_3D mode.

void pvSetModelPos(PVPoint p)

Sets the translation vector applied to 3D vertex when in PV_3D mode.

void pvSetModelPivot(PVPoint p)

Sets rotation pivot applied to 3D vertex when in PV_3D mode.

void pvSetDepthField(double front,double back)

Sets the front and back Z limits used when in PV_2D mode, mainly for hardware calibration.

void pvSetCull(pvCULL cull)

Sets the culling mode for polygons. Possible values are:

PV_CULL_NONE	No culling, every polygons are displayed
PV_CULL_CW	Cull polygons that are clockwise
PV_CULL_CCW	Cull polygons that are counter clockwise

void pvSetLightingMode(pvLIGHT mode)

Tells PV to performs lighting or not when in PV_3D mode. Possible values are:

PV_LIGHTING	PV will perform lighting according to vertex's normal
PV_NOLIGHTING	PV will not perform lighting, you should set light info with *pvSetColor()* and *pvSetSpecular()*

void pvAddLight(PVLight *l,unsigned *handle)

Adds a PVLight *l* object to the primitive subsystem, this light will be used in shading computation when in PV_3D and PV_LIGHTING mode. The function returns a handle for this light at the address specified by *handle*. The light *l* is not added to the subsystem but copied into it, as a result your original PVLight object (*l*) should be killed by an explicit call to *PV_KillLight()* this kill will not delete the copied light from the subsystem.

void pvRemoveLight(unsigned handle)

Removes a light referenced by handle *handle* from the primitve subsystem.

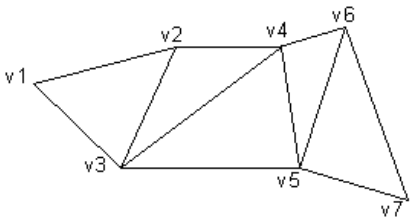
void pvSetAmbientLight(float r,float g,float b,float a)

Sets ambient light parameters.

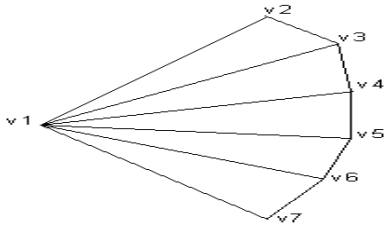
void pvBegin(pvPRIMIT prim,UPVD8 *Scr)

Begins a primitve drawing, the primitive type *prim* must be one of the following:

PV_TRIANGLES	The primitives will be composed of 3 vertices each time
PV_TRIANGLES_STRIP	Vertices form a triangle strip
PV_TRIANGLES_FAN	Vertices form a triangle fan
PV_POLYS	The primitives is an arbitrary convex polygon with up to 16 vertices
PV_QUADS	The primitive is a quadrilateral.



Triangle strip



Triangle fan

When drawing a triangle strip, the system uses vertices v1, v2, and v3 to draw the first triangle, v2, v4, and v3 to draw the second triangle, v3, v4, and v5 to draw the third, v4, v6, and v5 to draw the fourth, and so on.

When drawing a triangle fan, the system uses vertices v1, v2, and v3 to draw the first triangle, v3, v4, and v1 to draw the second triangle, v1, v4, and v5 to draw the third triangle, and so on.
The *scr* parameter have the same semantic than the one of *PV_RenderWorld()*.

void pvSetMaterial(PVMaterial *m)

Sets the PVMaterial used for rendering.

void pvMapCoord(float u,float v)

Sets u and v mapping coordinates for the current vertex.

void pvAmbMapCoord(float u,float v)

Sets ambient mapping coordinates for the current vertex.

void pvNormal(float x,float y,float z)

Sets the coordinates for the normal of the current vertex.

void pvColor(float r,float g,float b,float a)

Sets the color for the current vertex.

void pvSpecular(float r,float g,float b,float a)

Sets the specular for the current vertex.

void pvColorIndex(UPVD8 index)

Sets the color index of the current vertex (used when in PVM_PALETIZED8 mode to specify a color in the system palette).

void pvMapCoordv(float *uv)

Same than *pvMapCoord()* but loads parameters from an array of 2 float pointed by *uv*. Saves on memory bandwidth.

void pvAmbMapCoordv(float *uv)

Same than *pvAmbMapCoord()* but loads parameters from an array of 2 float pointed by *uv*. Saves on memory bandwidth.

void pvNormalv(float *xyz)

Same than *pvNormal()* but loads parameters from an array of 3 float pointed by *xyz*. Saves on memory bandwidth.

void pvColorv(float *rgba)

Same than *pvColor()* but loads parameters from an array of 4 float pointed by *rgba*. Saves on memory bandwidth.

void pvSpecularv(float *rgba)

Same than *pvSpecular()* but loads parameters from an array of 4 float pointed by *rgba*. Saves on memory bandwidth.

void pvVertexfv(float *xyz)

Same than *pvVertexf()* but loads parameters from an array of 3 float pointed by *xyz*. Save on memory bandwidth.

void pvVertexv(double *xyz)

Same than *pvVertex()* but loads parameters from an array of 3 double pointed by *xyz*. Save on memory bandwidth.

void pvSetWrapFlag(PVFLAGS flags)

Sets the flags that should be used for the forthcoming primitives they are the same than those accepted by PVFace.Flags.

void pvVertex(double x,double y,double z)

Loads a vertex to the subsystem. This call terminates a vertex, all colors mapping coordinates normal and so on for this vertex should have been specified before this call.
In PV_2D mode, *x* and *y* are the coordinates on the screen and *z* is the inverse of the Z, used for perspective correction and depth buffering (*pvSetDepthField()* should be set accordingly).
In PV_3D mode, *x*, *y* and *z* are the coordinates of the vertex.

void pvVertexf(float x,float y,float z)

Same than *pvVertex()* but with float parameters, to save on memory bandwidth.

void pvSetFogType(PVFogType type)

Sets the fog type for further rendered primitives. See *PVFogInfo* structure.

void pvSetFogStart(float start)

Sets the fog *start* parameter. See *PVFogInfo* structure.

void pvSetFogEnd(float end)

Sets the fog *end* parameter. See *PVFogInfo* structure.

void pvSetFogDensity(float density)

Sets the fog *density* parameter. See *PVFogInfo* structure.

void pvSetFogColor(float r,float g,float b)

Sets the fog color.

void pvEnableIndex(PVFLAGS flags)

Selects the data that will be transfered when a call to *pvVertexIndexedf()* is issued. *Flags* can be a combination of :

PPI_MAPCOORD	2 floats representing the mapping coordinates
PPI_AMBMAPCOORD	2 floats representing the ambient mapping coordinates
PPI_COLOR	4 floats representing the rgba color
PPI_SPECULAR	4 floats representing the rgba specular color
PPI_NORMAL	3 floats representing the normal
PPI_COLORINDEX	1 byte representing the color index

This permits to group in one *pvVertexIndexed()* call all the parameters needed to describe a vertex, gaining on memory bandwidth.

void pvSetVertexIndex(float *xyz0,unsigned stride)

Sets the data about vertex indexing. *xyz0* is the base of the array containing the X,Y,Z float values of each vertex. *stride* is the number of byte between two vertices.

void pvSetMapIndex(float *uv0,unsigned stride)

Sets the data about mapping coordinates indexing. *uv0* is the base of the array containing the U,V float values for each vertex. *stride* is the number of byte between two sets of coordinates.

void pvSetAmbMapIndex(float *auav0,unsigned stride)

Sets the data about ambient mapping coordinates indexing. *auav0* is the base of the array containing the AU,AV float values for each vertex. *stride* is the number of byte between two sets of coordinates.

void pvSetColorIndex(float *rgba0,unsigned stride)

Sets the data about color indexing. *rgba0* is the base of the array containing the R,G,B,A float values for each vertex. *stride* is the number of byte between two sets of coordinates.

void pvSetSpecularIndex(float *rgba0,unsigned stride)

Sets the data about specular color indexing. *rgba0* is the base of the array containing the R,G,B,A float values for each vertex. *stride* is the number of byte between two sets of coordinates.

void pvSetNormalIndex(float *xyz0,unsigned stride)

Sets the data about normal coordinates. *xyz0* is the base of the array containing the X,Y,Z float values for each vertex. *stride* is the number of byte between two sets of coordinates.

void pvSetColorIndexIndex(UPVD8 *c0,unsigned stride)

Sets the data about color index indexing. *c0* is the base of the array containing the C float values for each vertex. *stride* is the number of byte between two sets of coordinates.

void pvVertexIndexedf(unsigned index)

Loads *index*-th entry of all arrays selected by *pvEnableIndex()*.

void pvEnd(void)

Ends a primitive. If rendering a PV_POLYS primitive, closes the polygon.

SPLINES

Splines

Panard Vision offers facilities to handle some rationnal curves and surfaces. Two structures are defined, one to describe a curve, the other for surfaces. The splines are not limited in dimensions of their control points.

Struct PVSpline1

This structure holds informations about an n-D curve.

- *Order* : Number of control points
- *NbrComponent* : Number of components per control points, this also dictates the number of components that will be generated at evaluation time
- *Points* : Pointer to an array of float holding the control points.
- *u1, u2* : lower and higher value that will be used to map the parameter at evalution time (usually 0.0 and 1.0)

Struct PVSpline2

This structure holds informations about a n-D surface.

- *UOrder, Vorder* : Number of control points in U and V direction of the patch.
- *NbrComponent* : Number of components per control points, this also dictates the number of components that will be generated at evaluation time
- *Points* : Pointer to an array of float holding the control points.
- *u1, u2* : Lower and higher value that will be used to map the parameter *u* at evalution time (usually 0.0 and 1.0)
- *v1,v2* : Lower and higher value that will be used to map the parameter *v* at evalution time (usually 0.0 and 1.0)
- *CalcNormal* : If set to 1 then normal to the surface will be generated by the evaluation function
- *AutoNormalize* : If set to 1 then generated normal, will be normalized

PVSpline1 * PV_CreateSpline1(unsigned order,unsigned nbrcomponent)

Creates a spline curve object with *order* control points, and *nbrcomponent* components by control points.

PVSpline2 * PV_CreateSpline2(unsigned uorder,unsigned vorder,unsigned nbrcomponent)

Creates a spline surface object with *uorder* control points in u direction, *vorder* control points in *v* direction, and *nbrcomponent* components by control points.

void PV_KillSpline1(PVSpline1 *s)

Destroys a spline curve object.

void PV_KillSpline2(PVSpline2 *s)

Destroys a spline surface object.

void PV_EvalSpline1(PVSpline1 *pvs,float u,float result[])

Computes the position of a point on the spline curve *pvs* positionned at *u* on the curve. The value of *u* are mapped beetwen the *u1* and *u2* fields of *pvs*. The returned point has the same number of components than the control points of the curve and is stored in *result*.

void PV_EvalSpline2(PVSpline2 *pvs,float u, float v,float result[],PVPoint *Normal)

Computes the position of a point on the spline surface *pvs* positionned at *u,v* on the surface. The value of *u* and *v* are mapped beetwen the *u1,u2* and *v1,v2* fields of *pvs*. The returned point has the same number of components than the control points of the curve and is stored in *result*. If *pvs* has *CalcNormal* set to 1 then the normal at the computed point is stored in *Normal*.

ADVANCED USERS

Lightmaps

Lightmaps is a technic used to add very realist lighting effects to a world without cutting on framerate. This methos has been firstly used in a game with « Quake ».
The principle is very simple, at each face is associated a texture (called the lightmap) which has the same size than the face in texture space (ie there is a correspondance beetween the number of texel needed to fill the face and the size the lightmap). This texture represents the quantity of light received by each texel. The lightmap can then be computed with very cpu intensive algorithms (like raytracing or radiosity) offline, and then be used at runtime with no speed penalty. At render time, this lightmap is blended over the normal texture to achieve the light effect.
The main problem is that lightmaps are static.

To simulate some dynamic lighting effetcs, there can be numerous lightmap for a singlme face. The correct lightmap is switched at runtime to simulate by example, flashing lights, or light switch on/off. This is called pseudo-dynamic lightmaps.

Panard Vision supports either monochromatic lightmaps (like in Quake I) or RGB lightmaps (like in Quake II) giving more freedom but being more memory consuming.

To cut down on memory requirements, lightmaps can be undersampled. A lightmap is undersampled when one of his texel covers many texel of the face's texture. Undersampling are power of 2 downsampling. An undersampling of 3 means that one texel of the lightmap covers 8 texels of the face's texture.

Struct PVLightMap

This structure holds info about lightmaps. Each instance of this structure MUST be associated with a PVFace to be used. They cannot be shared among faces.

- *Flags :* Flags indicating the lightmaps attributes. The valid flags are :
  LIGHTMAP_RGB each lightmap texel is specified as a 24bit RGB value
  LIGHTMAP_MONO each lightmap texel is specified as 8 bit monochromatic value
- *Sampling* : Undersampling factor. 3 means that one texel of the lightmaps covers a 8*8 block of the face's texture.
- *NbrLightMaps* : Number of lightmaps for this face. This is used when using pseudo-dynamic lightmaps. There is numerous lightmaps for a single face. There can be up to MAX_LIGHTMAPS per face (defined in config.h)
- *CurrentLightMap :* Current lightmap to be ued for rendering. Used to switch beetween numerous pseudo dynamic lightmaps.
- *Width, Height :* Width and Height of the lightmaps (pseudo dynamic lightmaps have all the same size)
- *Maps[MAX_LIGHTMAPS] :* array of pointer to the lightmaps.

How to compute the size of a lightmap

The dimensiopns of a lightmap are directly linked to the size of face in texture space. For a given face, find the minimum and maximum for *u* and *v* texture coordinates. Multiply this by the size of the face's texture to obtain, then compute the size of the face in texture space by doing the substration beetween min and max values for u and v. Scale this to your sampling factor and then add 1.
Here is the mathematical formulation:

```
s=(1<<Sampling);

MinU=floor(minu*TextureWidth/s)*s
MinV=floor(minv* TextureHeight/*s)*s
MaxU=ceil(maxu*TextureWidth/s)*s
MaxV=ceil(maxv*TextureHeight/s)*s

FaceWidth=MaxU-MinU
FaceHeight=MaxV-MinV
```


LightMapWidth=((FaceWidth>>Sampling)+1)

Then to construct your lightmaps don't forget that a lightmap MUST be *LightMapSize* pixel wide.

int PV_InitTextureCache(unsigned size)

This function initializes the texture cache (used only when rendering in software mode). It allocates *size* bytes of memory that will be used to hold blend textures with their lightmaps. The more bigger the cache, the faster the rendering.

void PV_ResetTextureCache(void)

This function flushes the texture cache.

void PV_ReleaseTextureCache(void)

This function dealloactes the cache.

PVLightMap * PV_CreateLightMapInfo(void)

Creates a new PVLightMap structure. Return NULL if not enough memory.

void PV_KillLightMapInfo(PVLightMap *l)

Destroys th PVLightMap pointed by *b*.

void PV_AttachLightMap(PVFace *f,PVLightMap *b)

Attach a PVLightMap *b* struct to a face *f*. This face will then use the lightmaps described by *b* at render time (if LIGHTMAP flag specidied in his material).

UPVD8 * PV_GetCachedTexture(PVFace *f,unsigned MipMapNum,unsigned *Width,unsigned *Height)

This function retrieves a texture blended with a lightmap for the face *f*, for mipmap number *mipmapnum*. The width and height of the returned texture are stored in the integer pointed by *Width* and *Height*. The format of the texture returned is the same than the pixel format specified by *PV_SetRGBIndexingMode()* in RGB modes or a paletized texture using the *Global256Palette* palette of the world where the face lies.
This function is targeted at user defined rasterizers.

Faces explained

Structure PVFace

- *Material* : This is the name of the material that should be used for rendering, this member should be set by the client program. The name will be used during the compilation process to lookup materials in registered materials of the world. Once the lookup is done, the *Material* fields is freed/nulled and a pointer to the corresponding material is stored in *MaterialInfo* (see below). If you don't want to specify a name, but directly specify a pointer to a material, keep this field null, and fill *MaterialInfo* accordingly.
- *MaterialInfo* : This field is filled with a pointer to the material corresponding with the same name than specified in *Material*. Could be setup by client process if *Material* is null.
- *Filler* : Pointer to the filling routine. This is filled during the compilation process. But may be changed at runtime to use some fancy fillers.
- *Flags* : Rendering flags of the face. These are the same than the material rendering flags, this field is set during the compilation process. But you can specify some more like before the compilation or at the runtime:

U_WRAP	Tells PV that this face may be texture wrapped with respect to U, i.e. the texture coordinates should not be interpolated by the "inside" but by the "outside". Useful to fix problem using spherical mapping.
V_WRAP	Tells PV that this face may be texture wrapped with respect to U, i.e. the texture coordinates should not be interpolated

	by the “inside” but by the “outside”. Useful to fix problem using spherical mapping.
TEXTURE_BASIS	Tells PV to generate mapping coordinates using an orthogonal projection (using the *TextureBasis* info) of each vertex of the face. This way, vertices can have different mapping coordinates depending on the face being computed. See Quake reader sample.

- *Father* : Pointer to a PVMesh object where the face live. Automagically sets by *PV_SimpleCreateMesh()* or during compilation.
- *NbrVertices* : Number of vertices for this face, up to MAX_VERTICES_PER_POLY allowed. Must be set by the client process.
- *V* : array of index in the *Father* data array describing the vertices of the faces. Must be set by the client process.
- *Zavg* : Average depth of the face. Computed by PV during rendering, available for rasterization.
- *DistanceToCam* : minimum distance with respect to z of the face to the camera. Computed by PV during rendering, available for rasterization and clipping.
- *Poly* : pointer to the polygon resulting of the 3D clipping. May be null if not clipped. Handled by PV.
- *Normal* : Normal to face. Filled by a call to *PV_MeshNormCalc()*. May be filled by client process.
- *LightMap* : Pointer to PVLightMap structure. This hold the lightmaps associated with this face.

User defined lights

You can define your own lighting model by defining « user light ». A user light is a light with a type field of PVL_USER_LIGHT. When the lighting module of PV encounter such a light he calls the callback function pointed by *PV_UserLight* (see callbacks functions sections). It's then up to you to fill in the shading info for the mesh.

The Shading table

Each mesh has a member named *Shading* which is an array stating for each vertex a record like this

```
typedef struct _ShadingInfo
{
      UPVD8 Shade;
      PVRGBF Color;
      PRGBF Specular;
} PVShadeInfo;
```

The *Shade* member is for PVM_PALETIZED8 and PVM_RGB16 modes, it's an index between 0 and 255 (0-128 are 0 intensity) stating the light intensity. The *color* field is for PVM_RGB mode stating the color for this vertex. These values will be then directly used by the rasterization module.

The VertexVisible table

Each mesh has a byte array, which state for each vertex if he is visible (1) or not (0). With this, you can do efficient computation for your light, forgetting non-interesting vertex.

Example

Your lighting routine could be something like that :

```
for(i=0,vv=o->VertexVisible;i<o->NbrVertices; i++,vv++)
{
  if (*vv==0) continue;

// Do your fancy lighting here
}
```

NOTE: You can attach your own info about this light with the *UserData* field of each light.

User defined material types

If the defined rendering do not suit your needs. You can define your own material type. To do this, add the USER_FX flag to your material type. Then at *PV_CompileMeshes* call the callback function pointed by *PV_UserCompileMaterials* will be called for your material and the face being compiled.
You can now setup everything want but especially you must set the *Filler* member for this face. This is a pointer to the filling routine that will be used to render this face (i.e. your material) to screen.

The prototypes for a filling routine is:

*void Filler(PVFace *f)*

For Info on what is contained in a PVFace record, see pvision.h

At render time you can retrieve the material info for your face by using it's *MaterialInfo* member which is a pointer to your *PVMaterial*.

NOTE: You can use the *UserData* field of each material to hold specific data.

User defined visibility pipeline

The user may choose to bypass the default visibility pipeline for some meshes (for instance to render through a BSP or Portals oriented methods). This is done by setting the *UserPipe* member of the concerned PVMesh object to point on a user coded routine wich will do the visibility tests. The prototype for the function is:

*int (PVAPI * UserPipeline)(struct _PVWorld *w,struct _PVMesh *m)*

The *w* parameter is the world being computed and the *m* parameter is the mesh in this world being computed. The user routine is responsible for the following action:

- determine which faces are visibles (hint: *PV_GetFrustumFlags()* may be used to classify a bounding box with respect to the frustum)
- call the *PV_ClipFaceToFrustum()* for each of those face in order to clip/transform them and determine their visibility.
- Update the *Visible* and *NbrVisibles* members of *m* accordingly.
- Return a code which could be a combination of these:

MESH_RECOMPUTE_LIGHT	Light computations must be performed
MESH_FRUSTUM_CLIPPED	There is clipped face in the *Visible field*. PV will handle clipped faces.

For a sample, see the Quake driver.

Generic triangle rasterizer

The generic triangle filler is able to interpolate up to 9 increments (integer or float) providing high precision subpixel accuracy and efficient rasterization.
It also automatically supports for you, Zbuffering and Sbuffering.
In order to use it you must write 2 initialization routines. These 2 routines are grouped in a *FillerUserFunct* structure:

```
typedef struct _FillerUserFunct {
    int(*InitFunc1)(void);
    void(*InitFunc2)(void);
} FillerUserFunct;
```

The return value of the first function is used to either stop (!=0) or continue (==0) the rendering process.

Global variables used during rasterization

In the following sections, variable marked as « *User modifiable* » are setup during one of the two user init routines

HlineRoutine (User modifiable)

This variable point to user written filling routine. This function should draw a scan line from x1 to x2 at the position *TriFill_BufOfs+deb* in the frame buffer.

```
extern void (__cdecl *HLineRoutine)(unsigned, unsigned)
```

Should be initialized in one of the two user init routines.

GenFill_NbrInc (User modifiable)

Number of increment to interpolate using integers (<=9). Must be set in the first user init routine

GenFill_NbrIncF (User modifiable)

Number of increment to interpolate using floats (<=*GenFill_NbrInc*). Must be set in the first user init routine

GenFill_CurrentFace

Pointer to the PVFace being rendered.

GenFill_p1, GenFill_p2, GenFill_p3

These 3 variables hold the vertex number of the currently rendered face.

GenFill_GradientX[MAX_INCREMENT] (User modifiable)

This array holds floating point gradient in respect of x for each interpolated values. This array is first filled by the generic filler routine between the two user init functions. Could be modified in the second.

GenFill_GradientY[MAX_INCREMENT] (User modifiable)

This array holds floating point gradient in respect of y for each interpolated values. This array is first filled by the generic filler routine between the two user init functions. Could be modified in the second.

GenFill_iGradientX[MAX_INCREMENT] (User modifiable)

This array holds fixed point gradient in respect of x for each of the integer interpolated values. This array is first filled by the generic filler routine between the two user init functions. Could be modified in the second.

GenFill_InitialValues[3][MAX_INCREMENT] (User modifiable)

This array (float) holds for each point of the currently rendered face the initial values of each interpolated values. Must be filled by the first user defined init function.

GenFill_CurrentLeftVal[MAX_INCREMENT]

Current fixed point value of each of the integer interpolated values for the current scanline.

GenFill_CurrentLeftValF[MAX_INCREMENT]

Current floating point value of each of the floating point interpolated values for the current scanline.

GenFill_Height

Height (with clipping) of the currently rendered face. Decremented at each scanline.

GenFill_CurrentY

Current scanline in the frame buffer.

RGBAmbientLight

Pointer to a PVRGBF struct describing the ambient light of the world where the face belongs to.

ClipMinX, ClipMaxX, ClipMinY, ClipMaxY, ScanLength

Copy of the parameter passed to *PV_SetClipLimit.*

TriFill_BufOfs (User modifiable)

Pointer to a frame buffer where to render. This pointer is modified during rendering process, hence it must be refreshed before each call to the generic filler.
This variable should point to first scanline in the frame buffer. During each iteration of the generic filler, *TriFill_BufOfs* is incremented by *ScanLength*.

Functions available for rasterization

void GenFiller(PVFace *f,FillerUserFunct *FUT)

This is the generic filler routine. After setting up *TriFill_Ofs*, call it with a pointer to the face being rendered and your two init routines packed in a *FillerUserFunct* structure.

void GenFill_Wrap(unsigned i)

Wrap the i^{th} coordinates of the *InitialValues* array. Useful for coordinates textures on a sphere by example.

unsigned GetMipMap(unsigned nbrmips,float mu,float mv)

Returns the mipmap index of the texture required for drawing. *nbrmips* is the number of mipmaps of the material, *mu* end *mv* are the gradient in U and V used to compute the mipmap number. Use this function if you have linear gradients (i.e. no perspective correction).

unsigned GetMipMapIndex(unsigned nbrmips,float d,unsigned incu,unsigned incv)

Returns the mipmap number of the texture needed according to *nbrmips* number of mipmap level. *d* distance from the camera of the most close vertex of the face (use *DistanceToCam* member of PVFace). *incu* and *incv* are the index number in the *GenFill_Gradient*x arrays for the U and V coordinates.

Example : Affine texture mapper with mipmapping

User defined init routines

```
static int Init1Mapping(void) {
    PVMesh *o=GenFill_CurrentFace->Father;

    GenFill_NbrInc=2;

      // Mapping
        GenFill_InitialValues[0][0]=o->Mapping[GenFill_p1].u;
      GenFill_InitialValues[0][1]=o->Mapping[GenFill_p1].v;
      GenFill_InitialValues[1][0]=o->Mapping[GenFill_p2].u;
      GenFill_InitialValues[1][1]=o->Mapping[GenFill_p2].v;
      GenFill_InitialValues[2][0]=o->Mapping[GenFill_p3].u;
      GenFill_InitialValues[2][1]=o->Mapping[GenFill_p3].v;
```



```
      if(GenFill_CurrentFace->Flags&U_WRAP) GenFill_Wrap(0);
      if(GenFill_CurrentFace->Flags&V_WRAP) GenFill_Wrap(1);

return 0;
}

static void Init2Mapping(void) {
 float mu,mv;
 unsigned MipIndex;
 unsigned TextureW,TextureH;


    // MipMap
    if((PV_Mode&PVM_MIPMAPPING)&&(GenFill_CurrentFace->MaterialInfo-
>TextureFlags&TEXTURE_MIPMAP))
    {
        mu=fabs(GenFill_GradientX[0]*(GenFill_CurrentFace->MaterialInfo->Tex[0].Width-1));
        mv=fabs(GenFill_GradientX[1]*(GenFill_CurrentFace->MaterialInfo->Tex[0].Height-1));
        MipIndex=GetMipMap(GenFill_CurrentFace->MaterialInfo->NbrMipMaps,mu,mv);

        Texture=GenFill_CurrentFace->MaterialInfo->Tex[MipIndex].Texture;
        TextureW=GenFill_CurrentFace->MaterialInfo->Tex[MipIndex].Width;
        TextureH=GenFill_CurrentFace->MaterialInfo->Tex[MipIndex].Height;
    }
    else
    {
        // No MipMap
        Texture=GenFill_CurrentFace->MaterialInfo->Tex[0].Texture;
        TextureW=GenFill_CurrentFace->MaterialInfo->Tex[0].Width;
        TextureH=GenFill_CurrentFace->MaterialInfo->Tex[0].Height;
    }

    GenFill_GradientX[0]*=TextureW-1;
    GenFill_GradientX[1]*=TextureH-1;
    GenFill_GradientY[0]*=TextureW-1;
    GenFill_GradientY[1]*=TextureH-1;
    GenFill_iGradientX[0]=(int)(GenFill_GradientX[0]*65536.0);
    GenFill_iGradientX[1]=(int)(GenFill_GradientX[1]*65536.0);

    // MipMap correction
    GenFill_InitialValues[0][0]*=(TextureW-1);
    GenFill_InitialValues[1][0]*=(TextureW-1);
    GenFill_InitialValues[2][0]*=(TextureW-1);
    GenFill_InitialValues[0][1]*=(TextureH-1);
    GenFill_InitialValues[1][1]*=(TextureH-1);
    GenFill_InitialValues[2][1]*=(TextureH-1);

    HLineRoutine=HLineAffineMapping;
}

FillerUserFunct AffineMapping={Init1Mapping,Init2Mapping};
```

Call to the generic filler routine

```
TriFill_BufOfs=FrameBuffer ;
GenFiller(TheFace,&AffineMapping ;
```

The span routine

```
void __cdecl HLineAffineMapping(unsigned deb,unsigned fin)
{
    unsigned length;

    length=fin-deb;
    ofs=TriFill_BufOfs;
    ofs+=deb;
    while(length--!=0)
    {

*(ofs++)=Texture[(((GenFill_CurrentLeftVal[1]>>16)&AndHeight)<<ShiftWidth)+((GenFill_CurrentLe
ftVal[0]>>16)&AndWidth)];
        GenFill_CurrentLeftVal[0]+=GenFill_iGradientX[0];
        GenFill_CurrentLeftVal[1]+=GenFill_iGradientX[1];
    }
}
```


Generic perspective corrected span renderer

This span filler provides generic affine perspective correction four your routines. It uses the generic filler. The position 0 in all *InitialValues*, *Gradientxx* and so on arrays is reserved to store the 1/z values. But the 1/z value is discarded when calling the final span routine (i.e. all the values are shifted by one), that means that you could use the SAME routine with or without perspective correction for drawing your horizontal span. The generic perspective corrected span renderer will only works with floating point values (i.e. GenFill_NbrIncF value of the generic Triangle Rasterizer), and is able to provide floating point and integer values to your routines.
X86 architectures: if you use the generic span renderer, you MUST not use more than 7 FPU registers in your span filler routine. It's easy to check if your span routine is in asm, it's less easy when your span routine is in C. Be warned, if you got some strange results, check this out.

Global variables

GenPHLine_NbrIncF2F

Number of floating increment to keep as floating values for the span routine. Should be set during the first Init routine. For example you want floating point u and v, set GenPHLine_NbrIncF2F to 2.

GradientXAff[MAX_INCREMENT]

Affine gradients array. Must be filled in the second init routine by multiplying *GradientX* by AFFINE_LENGTH.

SlineRoutine

Pointer to a span filling routine. Usually point to the same filling routine you used for non corrected rendering.
Prototype is :

```
void (__cdecl *SLineRoutine)(unsigned, unsigned)
```

HLPInitFunc (x86 architecture only)

Pointer to a void function called before each call to the *SLineRoutine.*
Prototype is :

```
void (__cdecl *HLPInitFunc)(void);
```

Functions

GenPHLine

The generic perspective span renderer routine. Usually it is the one pointed by *HLineRoutine*.

PV_SetAutomaticPerspectiveRatio(float r)

Sets the ratio for automatic perspective correction decision. The ratio is the result of the closest z divided by the farthest one of the face.

PV_GetAutomaticPerspectiveRatio(float r)

Retrieves the value set by *PV_SetAutomaticPerspectiveRatio*.

PerspectiveNeeded(float z1,float z2,float z3)

Returns 1 if perspective correction is required according to the automatic ratio sets by *PV_SetAutomaticPerspectiveRatio*. z1, z2, z3 are the 3 values of the 3 vertices of the face.
Returns 0 otherwise.

void PV_SetPerspectiveMipBias(float bias)

Sets the agressivity of mipmapping on perspective corrected materials. The bigger *bias* the blurrier the image.

Example : Perspective corrected mapping with mipmap

User Init Routines

```
static int Init1Mapping(void) {
    PVMesh *o=GenFill_CurrentFace->Father;

    if(GenFill_CurrentFace->Flags&AUTOMATIC_PERSPECTIVE)
    if(!PerspectiveNeeded(o->Rotated[GenFill_p1].zf,o->Rotated[GenFill_p2].zf,o-
>Rotated[GenFill_p3].zf))
    {
        TriAffineMapping(GenFill_CurrentFace);
        return 1;
    }

    GenFill_NbrIncF=3;

    // 1/z
    GenFill_InitialValues[0][0]=o->Projected[GenFill_p1].InvertZ;
    GenFill_InitialValues[1][0]=o->Projected[GenFill_p2].InvertZ;
    GenFill_InitialValues[2][0]=o->Projected[GenFill_p3].InvertZ;

    // u, v
    GenFill_InitialValues[0][1]=o->Mapping[GenFill_p1].u;
    GenFill_InitialValues[0][2]=o->Mapping[GenFill_p1].v;
    GenFill_InitialValues[1][1]=o->Mapping[GenFill_p2].u;
    GenFill_InitialValues[1][2]=o->Mapping[GenFill_p2].v;
    GenFill_InitialValues[2][1]=o->Mapping[GenFill_p3].u;
    GenFill_InitialValues[2][2]=o->Mapping[GenFill_p3].v;

    if(GenFill_CurrentFace->Flags&U_WRAP) GenFill_Wrap(1);
    if(GenFill_CurrentFace->Flags&V_WRAP) GenFill_Wrap(2);

    // 1/z
    GenFill_InitialValues[0][1]*=GenFill_InitialValues[0][0];
    GenFill_InitialValues[0][2]*=GenFill_InitialValues[0][0];
    GenFill_InitialValues[1][1]*=GenFill_InitialValues[1][0];
    GenFill_InitialValues[1][2]*=GenFill_InitialValues[1][0];
    GenFill_InitialValues[2][1]*=GenFill_InitialValues[2][0];
    GenFill_InitialValues[2][2]*=GenFill_InitialValues[2][0];

    if((GenFill_CurrentFace->Flags&FLAT)||(GenFill_CurrentFace->Flags&GOURAUD))
    {
        col=GetRampIndex(GenFill_p1,o,GenFill_CurrentFace->MaterialInfo);
        col+=GetRampIndex(GenFill_p2,o,GenFill_CurrentFace->MaterialInfo);
        col+=GetRampIndex(GenFill_p3,o,GenFill_CurrentFace->MaterialInfo);
        col/=3;
        MapLightTranslateTable=GenFill_CurrentFace->MaterialInfo->PaletteTranscodeTable;
        MapLightTranslateTable16=GenFill_CurrentFace->MaterialInfo->RGB16TranscodeTable;
        LineMapLightTransTable=&MapLightTranslateTable[col*256];
        LineMapLightTransTable16=&MapLightTranslateTable16[col*256];
    }

    HLineRoutine=GenPHLine;

    SLineRoutine=&HLineAffineMapping;         // The same than before

    }

    return 0;
}

static void Init2Mapping(void) {
    unsigned MipIndex;
    unsigned TextureW,TextureH;

    // MipMap
    if((PV_Mode&PVM_MIPMAPPING)&&(GenFill_CurrentFace->MaterialInfo-
>TextureFlags&TEXTURE_MIPMAP))
    {
        MipIndex=GetMipMapIndex(GenFill_CurrentFace->MaterialInfo-
>NbrMipMaps,GenFill_CurrentFace->DistanceToCam,1,2);

        Texture=GenFill_CurrentFace->MaterialInfo->Tex[MipIndex].Texture;
        TextureW=GenFill_CurrentFace->MaterialInfo->Tex[MipIndex].Width;
        TextureH=GenFill_CurrentFace->MaterialInfo->Tex[MipIndex].Height;
```



```
}
    else
    {
        // No MipMap
        Texture=GenFill_CurrentFace->MaterialInfo->Tex[0].Texture;
        TextureW=GenFill_CurrentFace->MaterialInfo->Tex[0].Width;
        TextureH=GenFill_CurrentFace->MaterialInfo->Tex[0].Height;
}

    GenFill_InitialValues[0][1]*=(TextureW-1);
    GenFill_InitialValues[1][1]*=(TextureW-1);
    GenFill_InitialValues[2][1]*=(TextureW-1);
    GenFill_InitialValues[0][2]*=(TextureH-1);
    GenFill_InitialValues[1][2]*=(TextureH-1);
    GenFill_InitialValues[2][2]*=(TextureH-1);

    GenFill_GradientX[1]*=TextureW-1;
    GenFill_GradientX[2]*=TextureH-1;
    GenFill_GradientY[1]*=TextureW-1;
    GenFill_GradientY[2]*=TextureH-1;

    AndHeight=TextureH-1;
    AndWidth=TextureW-1;

    GradientXAff[0]=GenFill_GradientX[0]*AFFINE_LENGTH;
    GradientXAff[1]=GenFill_GradientX[1]*AFFINE_LENGTH;
    GradientXAff[2]=GenFill_GradientX[2]*AFFINE_LENGTH;

}

FillerUserFunct PerspectiveMapping={Init1Mapping,Init2Mapping};
```

Call to the generic filler routine

```
TriFill_BufOfs=FrameBuffer ;
GenFiller(TheFace,&PerspectiveMapping ;
```

The span routine

The same than before. It means you can in no time, with the same code add perspective correction to your filler. Wooow that’s so cool!

Zbuffering

Panard vision performs Zbuffering by interpolating 1/z, you can access the software zbuffer through the functions below (hardware Zbuffer are a little less friendly, see hardware section). Values stored in the Zbuffer are IEE24 bit precision (C's float).

void PV_ClearZBuffer(void)

Clear the software Zbuffer using the currently set zbuffer clear routine.

float *PV_GetZBuffer(void)

Returns a pointer to the software Zbuffer. The Zbuffer is an array of float with the same size than the clip window.

void PV_SetZBufferClearRoutine(void (PVAPI * ZBufferRoutine)(void))

Set the routine that will be used to clear the Zbuffer. This can be used to use some fancy clear scheme.

void PV_ClearZBufferNormal(void)

Default zbuffer clear routine of PV.

File drivers for Panard Vision

For example on how to build a mesh driver, see 3dsread.c.
It's a 3Dstudio file reader:

Int LoadMeshFrom3DS(char *name,PVWorld *world)

Creates mesh in *world* according to data contained in the 3DS file *name*.

Portals

Portals can be seen as a more flexible alternative to BSP trees for in-door environment. They provide very efficient occlusion culling, with zero overdraw (in either software or hardware). Only what is seen is rendered.

PV can calculate the geometry uncovered by a portal and cull out all other geometry hidden. A portal is any connection between different areas of the visual database (see figure 1). Portals can be any shapes, i.e. a door or a window. PV will in real-time decides which part of the next room is visible through the portal and culls out all other geometry (see figure 2). Portals can be visible through other portals so that you can have multiple rooms seen through multiple portals (see figure 3).

How portals integrates in the hierarchical world graph

A portal is represented by a special PVMesh object (created by *PV_CreateMeshPortal()*). This mesh has only one face, which is the portal. A portal can have up to 16 vertices. If at rendering time this polygon is visible all children of the portal-mesh will be clipped into this portal polygon.
See below for how a world graph looks like when using portals.
One important thing to remember is that the world graph traversal should begin with the mesh where the camera is inside. PV can handle this if the camera has the CAMERA_AUTOLOCATE flag (but you can specify yourself where you want to start), the starting mesh will be chosen automatically by PV.

Portals are the only exception to the no-cycle-in-world-graph. A child of a portal may be an already in-hierarchy mesh. This is useful when a place can be seen from 2 different places (see figure 1 room D).

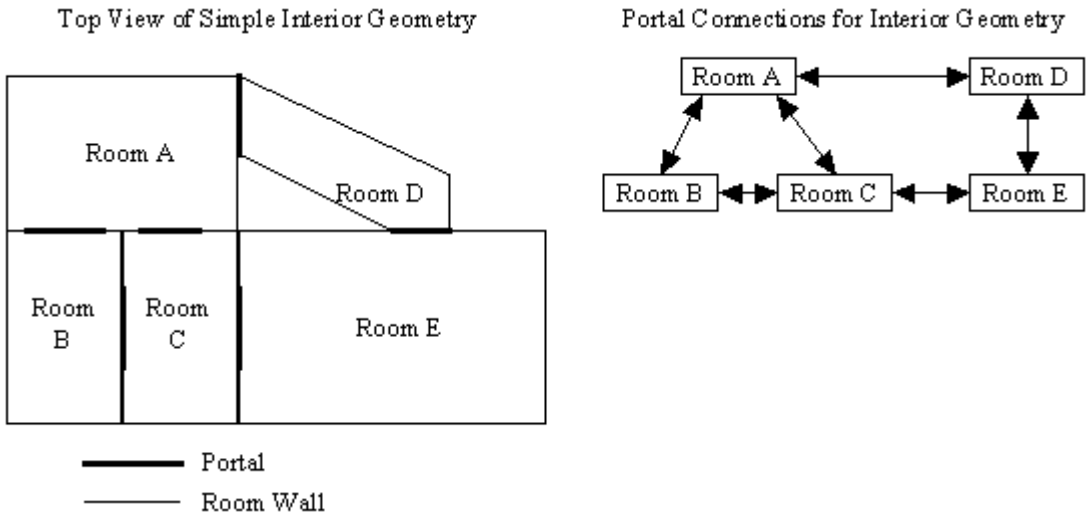
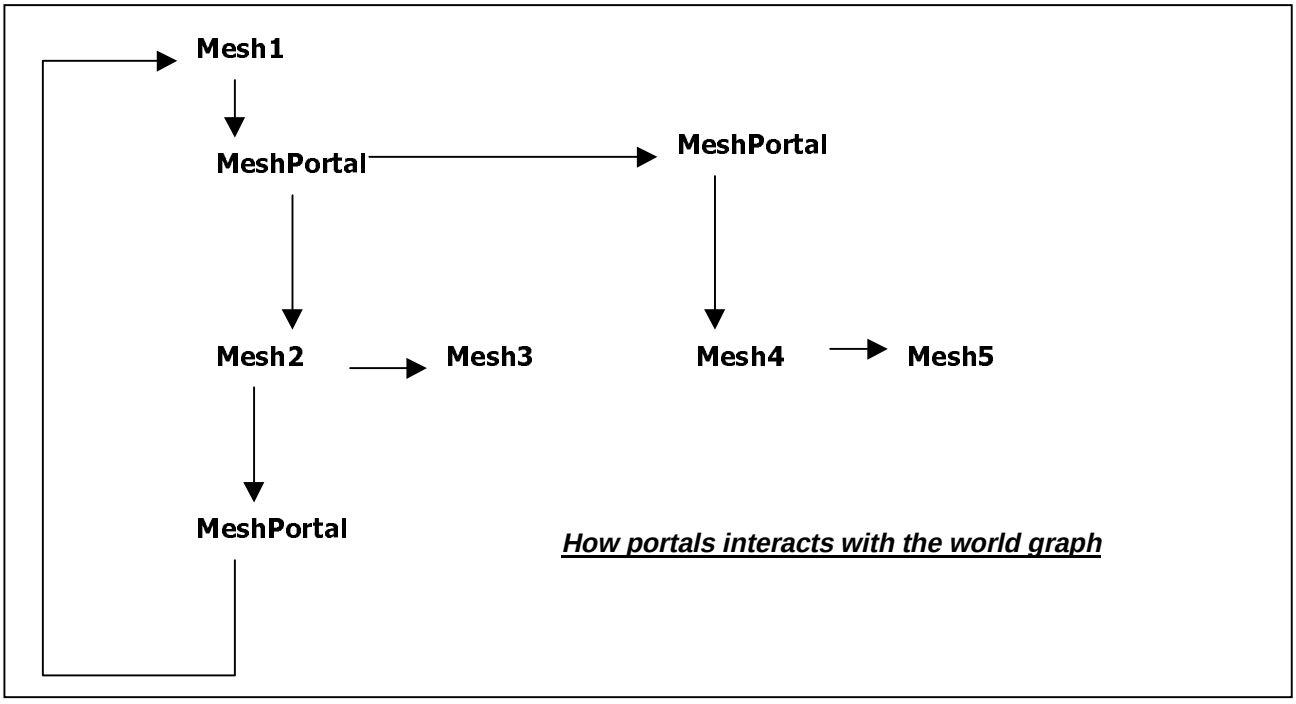
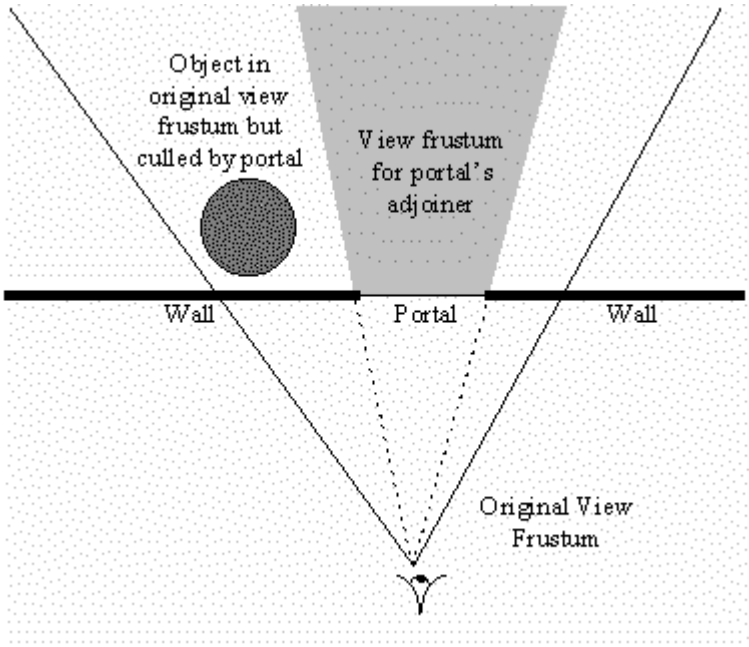
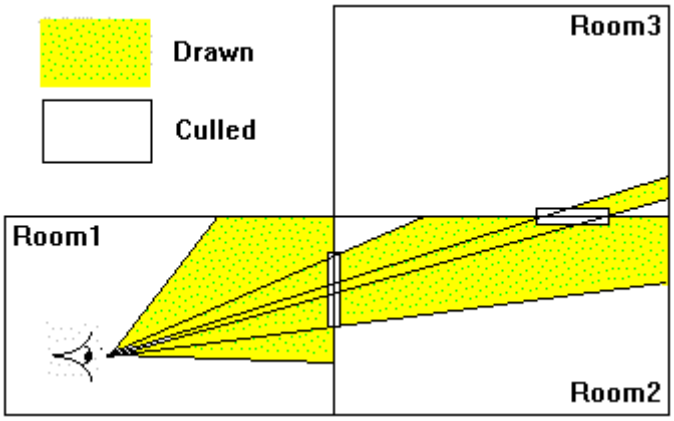


Figure 1 : A scene with link beetween rooms highlighted.







HARDWARE MANAGEMENT

Open 3D hardware API

See included drivers source for how to build a driver for Panard Vision.
In all subsequent functions, the *surfacenum* parameter refers to the one of the color buffer available on the hardware. However, the buffer numbered 0 will always be the default drawing buffer (front or back buffer depending on the hardware caps and driver caps). So direct access to specific buffer begins with the *surfacenum* 1.

To setup hardware rendering just set a hardware driver with *PV_SetHardwareDriver()* and then issue a *PV_SetMode(YourRenderMode|PVM_USEHARDWARE)*. If no error occured, PV will render through the choosen hardware. See samples for details.

NOTE: Some drivers may choose to ignore the *surfacenum* parameter.

Flags definition for hardware caps

```
// General Flags
#define PHG_ZBUFFER                     1       // Hardware supports ZBUffering
#define PHG_ALPHABUFFER                 2       // Hardware supprots alpha buffer
#define PHG_ALPHATEST                   4       // Hardware supports alpha testing against a reference alpha value

// Frame Access Flags
#define PHF_FRAMEBUFFER                 1       // The hardware supports Direct frame buffer access
#define PHF_DEPTHBUFFER                 2       // The hardware supports Direct depth buffer access
#define PHF_ALPHABUFFER                 4       // The hardware supports Direct alpha buffer access
#define PHF_BITBLTFB                    8       // The driver supports the BitBltFB function
#define PHF_BITBLTDB                    16      // The driver supports the BitBltDB function
#define PHF_BITBLTAB                    32      // The driver supports the BitBltAB function

// Texturing flags
#define PHT_BILINEAR                    1       // Hardware supports Bilinear filtering
#define PHT_MIPMAP                      2       // Hardware supports Mipmapping
#define PHT_TRILINEAR                   4       // Hardware supports trilinear filtering

// Blending Flags
#define PHB_SRCRGBBLEND                 1       // These flags indicates which fields are supported
#define PHB_DSTRGBBLEND                 2       // in material blending
#define PHB_SRCALPHABLEND               4
#define PHB_DSTALPHABLEND               8

// Format Caps
#define PHD_DEPTHPROP                   1       // Format of the depth buffer is propriatry
#define PHD_DEPTH16                     2       // depth values are 16 bits (fixed)
#define PHD_DEPTHFLOAT                  4

// Constants needed to define acces to the FrameBuffer
#define PFB_READONLY                    1       // Give a lock for reading only
#define PFB_WRITEONLY                   2       // Give a lock for writing only
```

Structure PVHardwareCaps

This structure describes the current hardware capabilities.

```
typedef struct _PVHardwareCaps {
        PVFLAGS GeneralCaps;
        PVFLAGS FrameCaps;
        PVFLAGS TextureCaps;
        PVFLAGS BlendCaps;
        PVFLAGS FormatCaps;

        unsigned BitsPerPixel;                          // Number of bits per pixel in the frame buffer
        unsigned NbrBitsRed,NbrBitsGreen,NbrBitsBlue,NbrBitsAlpha;  // Size of each components
        unsigned RedPos,GreenPos,BluePos,AlphaPos;      // Position of each color component in the pixel
        int  RowStride;                                 // Length of a framebuffer scanline in bytes

        int PhysicalResx,PhysicalResY;                  // Physical resolution of the screen
        unsigned NbrSurf;                               // Number of surfaces available

        unsigned BitsPerDepth;                          // Number of bits in a depth value red from the depth buffer

        unsigned BitsPerAlpha;                          // Number of bits in an alpha value red from the alpha buffer
} PVHardwareCaps;
```

Structure PVHardwareDriver

This structure defines the entry points of a valid driver.

```
typedef struct _PVHardwareDriver {
        unsigned Version;
        unsigned SizeOfCaps;                            // Size of the driver's PVHardwareCaps
        PVFLAGS Caps;                                   // Copied as the new pipeline control word, when driver is first initialized
        char *Desc;
                // Name of the driver
        int (PVAPI * Detect)(void);                     // Are you there method, !=0 if succesful detect
```



```
        int (PVAPI * InitSupport)(long i);                          // Init hardware, return COOL if no error
        void (PVAPI * EndSupport)(void);                            // unInit Hardware
        int (PVAPI * SetViewPort)(unsigned int cmx,unsigned int cMx,unsigned int cmy,unsigned int cMy); // Set drawing area if supported by hardware
        int (PVAPI * LoadTexture)(PVMaterial *m);                   // Loads a material to hardware
        void (PVAPI * DeleteTexture)(PVMaterial *m);                // Frees a previously downloaded texture
        void *(PVAPI *GetFiller)(PVFLAGS flags);                    // Gets a pointer to a filler according to the flags
        int (PVAPI * PreRender)(PVWorld *w,unsigned surafecnum,PVFLAGS PVMode,PVFLAGS PVPipe);       // Called before the effective renering starts
        void (PVAPI * PrepareFace)(PVFace *f);                      // Called before each face
        void (PVAPI * PostRender)(void);                            // Called at the end of the rendering
        void (PVAPI * BeginFrame)(PVFLAGS pvmode,PVFLAGS pvpipe);           // Mark the beginning of a scene
        void (PVAPI * EndFrame)(void);                                                                          // Guess
        int (PVAPI * FlipSurface)(void);                            // Swap Buffers (for n-buffering hardware)
        int (PVAPI * FillSurface)(unsigned surfacenum,float r,float g,float b,float a);     // Fill the surfacenum buffer with the specified color
        void (PVAPI * RefreshMaterial)(PVMaterial *m);              //  Update the material state (not the texture content !)

        PVHardwareCaps *(PVAPI * GetInfo)(void);                    // Retrieve driver/hardware capabilities
        void *(PVAPI * LockFB)(unsigned surfacenum,PVFLAGS mode);   // Lock the frame buffer, returns a pointer to the frame buffer
        void (PVAPI * UnLockFB)(unsigned surfacenum,PVFLAGS mode);// Unlock a previously locked frame buffer
        void *(PVAPI * LockDB)(PVFLAGS mode);                       // Lock the depth buffer, returns a pointer to the depth buffer
        void (PVAPI * UnLockDB)(PVFLAGS mode);                      // Unlock a previously locked depth buffer
        void *(PVAPI * LockAB)(PVFLAGS mode);                       // Lock the alpha buffer, returns a pointer to the alpha buffer
        void (PVAPI * UnLockAB)(PVFLAGS mode);                      // Unlock a previously locked alpha buffer
        int (PVAPI * BitBltFB)(unsigned surfacenum,unsigned xdst,unsigned ydst,void *src,unsigned width, unsigned height,unsigned stride);
// BitBlt a rectangular arrys direcly to the frame buffer (source should be at the format reported by getinfo)
        int (PVAPI * BitBltDB)(unsigned xdst,unsigned ydst,void *src,unsigned width, unsigned height,unsigned stride);
// BitBlt a rectangular arrys direcly to the depth buffer (source should be at the format reported by getinfo)
        int (PVAPI * BitBltAB)(unsigned xdst,unsigned ydst,void *src,unsigned width, unsigned height,unsigned stride);
// BitBlt a rectangular arrys direcly to the alpha buffer (source should be at the format reported by getinfo)
} PVHardwareDriver;
```

int PV_SetHardwareDriver(PVHardwareDriver *hd)

Set the hardware driver used by PV to *hd*.

PVHardwareDriver *PV_GetHardwareDriver(void)

Returns the currently selected driver.

int PV_InitAccelSupport(long i)

Start acceleration support. It may means the switch to full screen mode by example, must be called before every other rendering primitive. The parameter *i* is driver dependent.

void PV_EndAccelSupport(void)

Stops hardware support. Should be called at the end of your program to restore resources and normal working mode of the hardware.

void PV_RefreshMaterial(PVMaterial *m)

Tells hardware driver that the properties of the material have been changed. You should call this each time you change something in a material when using hardware. This function DOES NOT work for changing/reloading textures. Use different materials to do this.

int PV_FlipSurface(void)

Flip front and back buffer if the hardware support double buffering. Do nothing otherwise.

int PV_FillSurface(unsigned surfacenum,float r,float g,float b,float a)

Fills the surface *surfacenum* with the color defined by *r,g,b,a*.

void PV_GetHardwareInfo(PVHardwareCaps *caps)

This functions fills a **preallocated** (by client) PVHardwareCaps structure, with the current state of the driver. You should call this routine after every lock routines (see below) to know how to handle data red or to be written .

void *PV_LockFrameBuffer(unsigned surfacenum,PVFLAGS mode)

Returns a pointer to the color buffer requested. Could be NULL. The *mode* parameter should be one of the following :

`PFB_READONLY`	Gets a lock for Reading
`PFB_WRITEONLY`	Gets a lock for writing.

void PV_UnLockFrameBuffer(unsigned surfacenum,PVFLAGS mode)

Unlock a previously locked color frame buffer. Parameters are the same than the associated lock function.

void *PV_LockDepthBuffer(PVFLAGS mode)

Returns a pointer to the depth buffer. Could be NULL. The *mode* parameter should be one of the following :

PFB_READONLY	Gets a lock for Reading
PFB_WRITEONLY	Gets a lock for writing.

Void PV_UnLockDepthBuffer(PVFLAGS mode)

Unlock a previously locked depth frame buffer. Parameters are the same than the associated lock function.

void * PV_LockAlphaBuffer(PVFLAGS mode)

Returns a pointer to the alpha buffer. Could be NULL. The *mode* parameter should be one of the following :

PFB_READONLY	Gets a lock for Reading
PFB_WRITEONLY	Gets a lock for writing.

void PV_UnLockAlphaBuffer(PVFLAGS mode)

Unlock a previously locked alpha buffer. Parameters are the same than the associated lock function.

int PV_BitBltFrameBuffer(unsigned surfacenum,unsigned xdest,unsigned ydest,void *src,unsigned width, unsigned height,unsigned stride)

Copy a block of pixel from client memory to color buffer. *xdest* and *ydest* specify the target position in hardware memory of the block. *src* is a pointer to the source block in the client memory. *width* and *height* are size of the block. s*tride* is the size in bytes between two scanlines of the block.
The source block must be at a 8.8.8.8 format, i.e. 8 bit red, 8 bit green, 8 bit blue, 8 bit alpha (in this order).

int PV_BitBltDepthBuffer(unsigned xdest,unsigned ydest,void *src,unsigned width, unsigned height,unsigned stride)

Copy a block of values from client memory to depth buffer. *xdest* and *ydest* specify the target position in hardware memory of the block. *src* is a pointer to the source block in the client memory. *width* and *height* are size of the block. s*tride* is the size in bytes between two scanlines of the block.
The source block format depends on hardware and should be obtained with *PV_GetHardwareInfo().*

int PV_BitBltAlphaBuffer(unsigned xdest,unsigned ydest,void *src,unsigned width, unsigned height,unsigned stride)

Copy a block of values from client memory to alpha buffer. *xdest* and *ydest* specify the target position in hardware memory of the block. *src* is a pointer to the source block in the client memory. *width* and *height* are size of the block. s*tride* is the size in bytes between two scanlines of the block.
Each alpha value of the source must be 8 bit.

PANARD MOTION

Panard Motion

Panard Motion is a basic hierarchical animation system built on top of the Panard Vision library.
PM use a hierarchical tree holding the animation info. It handles translation, and rotation interpolations using quaternions. Every rotation must be expressed in quaternions.
It has been designed to replay static animation. However, you can use it for dynamic animations by setting only one key for each node and real-time changing this node.

Structure PMotion

This structure defines the root of an animation tree. Fields are :

- *CurrentTime* : a floating value indicating the position in animation in an arbitrary unit.
- *MaxTime* : The overall time the animation lasts.

PMotion *PM_CreateTree(void)

Creates an empty animation tree and returns a pointer to it.

void PM_KillTree(PMotion *m)

Free memory used by the animation tree.

void PM_SetCurrentTime(PMotion *m,float t)

Sets the field *CurrentTime* of *m* to *t.*

void PM_ComputeCurrentTime(PMotion *m)

Compute meshes position for the current time.

File drivers for Panard motion

For example on how to build an animation driver, see 3dspm.c.
It's a 3Dstudio file reader:

int LoadAnimFrom3DS(char *name,PMotion *m,PVWorld *w)

Load animation info from *name*, store them to the PMotion tree *m* and link PMotion nodes to corresponding meshes from *w* (matching names).

Speed advises

In order to have the best performances consider the following rules :

- Keep your textures as small as possible
- Mix several textures in a bigger 256*256 one by example to enhance cache efficiency
- Don't use perspective correction if it's not really needed (check your program with and without it)
- Flat shading is not bad, and can be very efficient in many ways (especially on big faces), think about it.
- True RGB mode could be faster than fake RGB ☺.
- Keep your mesh as small as possible
- Use prelit textures, NOTHING|MAPPING mode could be very beautiful and efficient
- Use level of details for your meshes (use the Name capability of the engine)
- Bilinear is not really efficient, maybe on a Cray. But what is possible is to turn off bilinear when moving, and turn it on when static. I.e. your character is moving with unfiltered texture mapping, and when it stops you turn on bilinear. You can try the AUTOMATIC_BILINEAR flag too.
- On hardware, think about using PVW_MATERIALS sorting methods, on a driver aware of this there may be a big gain in performance.
- If you design a hardware driver, think about supporting the PVP_NO_TRIANGULATE flag.
- You can render convex meshes without sorting them. So think about convexifying your meshes and use the MESH_NOSORT flag.
- When creating a mesh try using polygons instead of just triangles, processing will be faster.
- When using Panard Primitives 3D, try to maximize the work done between a pvBegin() and a pvEnd().
- When using Panard Primitives try to use strips, fan quads or whatever to reduce the number of pvVertex() calls needed to draw your stuff.

